

HARDWARE IMPLEMENTATION OF A
PIPELINED TURBO DECODER

GUAN WANG

Hardware Implementation of a Pipelined Turbo Decoder

By

©Guan Wang

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Engineering
Faculty of Engineering & Applied Science
Memorial University of Newfoundland

May 2010

St. John's

Newfoundland

Abstract

Turbo codes have been widely studied since they were first proposed in 1993 by *Berrou*, *Glavieux*, and *Thitimajshima* in “*Near Shannon Limit error-correcting coding and decoding: Turbo-codes*” [1]. They have the advantage of providing a low *bit error rate* (BER) in decoding, and outperform linear block and convolutional codes in low *signal-to-noise-ratio* (SNR) environments. The decoding performance of turbo codes can be very close to the *Shannon Limit*, about 0.7*decibel* (dB). It is determined by the architectures of the constituent encoders and interleaver, but is bounded in high SNRs by an error floor. Turbo codes are widely used in communications. We explore the codeword weight spectrum properties that contribute to their excellent performance. Furthermore, the decoding performance is analyzed and compared with the free distance asymptotic performance. A 16-state turbo decoder is implemented using *VHSIC Hardware Description Language* (VHDL) and then mapped onto a *field-programmable gate array* (FPGA) board. The hardware implementations are compared with the software simulations to verify the decoding correctness. A pipelined architecture is then implemented which significantly reduces the decoding latency.

Keywords: turbo codes, decoding performance, Monte Carlo simulations, FPGA implementation

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Dr. Ramachandran Venkatesan and Dr. Paul Gillard. Dr. Ramachandran Venkatesan and Dr. Paul Gillard gave me great support in the research, such as providing funding and the devices in the experiments, directing simulations, discussing results, correcting errors, advising on the research, and revising related paper and this thesis. They greatly helped me to be committed to my research, and finally finish this thesis.

Together, I thank my parents. They helped me to start my life in Canada, and encouraged me throughout my studies in the Memorial University of Newfoundland. I also thank my friends, Shi Chen, Dianyong Zhang, Lin Wu, Rui He, Tao Bian, Qiyun Zhang, and Alaa, for their constructive criticisms, both technical and non-technical, which helped me to better understand and improve myself.

Contents

Abstract	ii
Acknowledgements	iii
Contents	iv
List of Figures	vii
List of Abbreviations	x
1 Introduction	1
1.1 Background	1
1.2 Motivation for the Research	4
2 Principles of Turbo Codes	6
2.1 The Architecture of Turbo Encoders	6
2.1.1 RSC Encoders and Turbo Encoders	6
2.1.2 The AWGN Channel Simulation	9
2.2 The Architecture of Turbo Decoders	11
2.2.1 The MAP Algorithm	11
2.2.2 Iterative Decoding	15
2.2.3 Interleaver Design	17
2.2.4 Sliding Window Technique	21

2.3	Performance Analysis for Turbo Codes in the AWGN Channel	23
2.3.1	Distance Properties of Codewords	23
2.3.2	Performance Bounds for Convolutional Codes	25
2.3.3	Free Distance Asymptotic Performance of Turbo Codes	27
2.4	Summary	28
3	Monte Carlo Simulations of Turbo Codes	29
3.1	Monte Carlo Methods	29
3.2	Performance Investigation of Turbo Decoders	31
3.2.1	Decoding Performance of a 16-state Turbo Code	31
3.2.2	Calculating the Free Distance Asymptotic Performance	33
3.2.3	Free Distance Asymptotic Performance	35
3.3	Limited Bits Representing a Number	37
3.3.1	Limited Bits Representing a Number without Sliding Window	39
3.3.2	Limited Bits Representing a Number with Sliding Window	41
3.4	Summary	45
4	Hardware Implementation of a Pipelined Turbo Decoder	46
4.1	Xilinx XUPV5-LX110T Evaluation Platform	48
4.2	Hardware Considerations of Turbo Codes	50
4.3	FPGA Implementation of a Pipelined Turbo Decoder	53
4.3.1	Implementation of the MAX-LOG MAP algorithm	53
4.3.2	Implementation of a Turbo Decoder with the Sliding Window	59
4.3.3	The Implementation of a Pipelined Turbo Decoder	69
4.3.4	Conclusions	79
4.4	Hardware Implementation in Cheaper FPGA Devices	85
4.5	Summary	86
5	Conclusions and Future Work	87

5.1	Conclusions	87
5.2	Future Work	90
	References	91

List of Figures

2.1	A rate 1/3 turbo encoder	7
2.2	Noise histogram comparison	10
2.3	An iterative turbo decoder with the MAP algorithm	15
2.4	The structure of a block interleaver	17
2.5	Parameters k_1 and k_2 for the specified information block lengths	19
2.6	A sliding window technique	22
3.1	A 16-state turbo encoder	32
3.2	Decoding performance of a turbo code	33
3.3	Decoding performance and free distance asymptotic performance of the Berrou's example	36
3.4	Decoding performance and free distance asymptotic performance of the CCSDS encoder	37
3.5	Decoding Performance with 8 integer bits, without the sliding window	40
3.6	Decoding Performance with 9 integer bits, without the sliding window	40
3.7	Decoding performance with a 32-bit sliding window, 9 integer bits and 1 fraction bit . .	42
3.8	Decoding performance with a 48-bit sliding window, 9 integer bits and 1 fraction bit . .	43
3.9	Decoding performance with a 64-bit sliding window, 9 integer bits and 1 fraction bit . .	43
3.10	Decoding performance with a 80-bit sliding window, 9 integer bits and 1 fraction bit . .	44
3.11	Decoding performance with a 128-bit sliding window, 9 integer bits and 1 fraction bit .	44
4.1	XUPV5-LX110T FPGA board	48
4.2	A true dual-port block RAM	49
4.3	Virtex-5 DSP48E slice	50

4.4	Verification strategy	52
4.5	Computation complexities of different decoding algorithms	54
4.6	Architecture of the MAP/LOG-MAP/MAX-LOG-MAP algorithm	54
4.7	Architecture of ACSU in the MAX-LOG-MAP algorithm	55
4.8	Architecture of ACSU in the LOG-MAP algorithm	56
4.9	The dataflow of the MAX-LOG-MAP algorithm	56
4.10	the RAMB18 group	57
4.11	Results of the MAX-LOG-MAP algorithm in Java simulations	58
4.12	Waveforms of the MAX-LOG-MAP algorithm in VHDL implementation (1)	59
4.13	Waveforms of the MAX-LOG-MAP algorithm in VHDL implementation (2)	60
4.14	Device utilization summary of the MAX-LOG-MAP algorithm	61
4.15	Architecture of a turbo decoder (1)	62
4.16	Architecture of a turbo decoder (2)	63
4.17	The block diagram of a turbo decoder with a sliding window	64
4.18	The state diagram of the turbo decoder	65
4.19	The state diagram of the modified turbo decoder	67
4.20	Results of the turbo decoding in Java simulations	68
4.21	Waveforms (the first 4 bits in the 1st window) of the turbo decoder in VHDL imple- mentation	69
4.22	Waveforms (the decoded bit 0-3 of the last window) of the turbo decoder in VHDL implementation	70
4.23	Waveforms (the decoded bit 4-7 of the last window) of the turbo decoder in VHDL implementation	71
4.24	The decoding time of the turbo decoder	72
4.25	The block diagram of a pipelined turbo decoder with a sliding window	73
4.26	The timing in the MAX-LOG-MAP algorithm	74
4.27	The main controller in the pipelined turbo decoder	76
4.28	The beta controller in the pipelined turbo decoder	77

4.29	The alpha controller in the pipelined turbo decoder	78
4.30	Display in a hyper-terminal Window (Last 16 output signals in the last window)	79
4.31	The state of "000020"	80
4.32	The state of "100000"	81
4.33	Decoding time of the pipelined turbo decoder	82
4.34	Device utilization summary of the pipelined turbo decoder	83
4.35	Comparison with the Xilinx turbo decoders	84

List of Abbreviations and Acronyms

ACSU	– addition-comparison-selection unit
APP L-value	– a posteriori probability likelihood values
AWGN	– additive white Gaussian noise
$A(X)$	– WEF
$A(W, X)$	– IOWEF
BER	– bit error rate
BPSK	– binary phase shift keying
BSC	– binary symmetric channel
$B(X)$	– bit WEF
CCSDS	– Consultative Committee for Space Data Systems
CDF	– column distance function
CDMA	– code division multiple access
CWEF	– conditional weight enumerating function
d	– number of bits released in a window
d_i	– CDF
d_{free}	– free distance
DVB	– digital video broadcasting
FPGA	– field-programmable gate array
FSM	– finite state machine
IOWEF	– input-output weight enumerating function
L	– APP L-value
ℓ	– stage index
L_a	– a priori probability
L_c	– channel reliability
LDPC	– low density parity check
L_e	– extrinsic probability
LLR	– log likelihood ratio

LSB – least significant bit
 LUT – look-up table
 JPL – Jet Propulsion Laboratory
 k – iteration number
 k_1, k_2 – interleaver parameters
 M – encoder's memory length
 MAP – maximum a posteriori probability
 N – block/interleaver size
 N_{free} – number of information sequences causing free-distance codewords
 NASA – National Aeronautics and Space Administration
 P_b – free distance asymptotic performance
 $P_b(E)$ – bit error probability by first error events
 PCCC – parallel concatenated convolutional code
 $P_d(E)$ – word error probability by first error events with the weight d
 $P_f(E)$ – word error probability by first error events
 PSD – power spectrum density
 R – code rate
 r – received signal in the end of channel
 R_0 – cut-off rate
 RAM – random access memory
 RISC – reduced instruction set computer
 RSC – recursive systematic convolutional
 s – ending state
 s' – starting state
 SISO – soft-in and soft-out
 SNR – signal-to-noise ratio
 SOVA – soft output Viterbi algorithm
 u – information bit/sequence

UCF – user constraint file

u_ℓ – information bit at stage ℓ

\hat{u}_ℓ – estimated bit at stage ℓ

v – codewords/symbols

v_0, v_1, v_2 – a codeword/symbol corresponding to an information bit

VHDL – VHSIC hardware description language

VHSIC – very-high-speed integrated circuit

VLSI – very large scale integrated circuit

WCDMA – wide-band code division multiple access

WEF – weight enumerating function

WER – word error rate

w – window size

\tilde{w}_{free} – average Hamming weight of the free distance information sequences

y, y^1, y^2 – codewords

Chapter 1

Introduction

1.1 Background

Turbo codes are high performance coding schemes in the error control coding that is in the field of information theory. They were first introduced by *Berrou, Glavieux* and *Thitimajshima* in 1993, “*Near Shannon Limit error-correcting coding and decoding: Turbo-codes*”, published in *the Proceedings of IEEE International Communications Conference* [1]. Turbo coding achieved immediate worldwide attention. The importance of turbo codes is the fact that they enable reliable communications with power efficiencies close to the theoretical limit established by *Claude Shannon* [2]. They represent effective error control coding schemes in error-control coding theory. Since 1993, a large number of papers have been published about the performance of turbo codes. It is found that the excellent performance of turbo codes is determined by its encoder architecture, especially the employment of the interleavers, *recursive systematic convolutional* (RSC) encoders, and the iterative decoding process. Consequently, the design methodology is widely studied.

The advantage of turbo codes is that they outperform other coding schemes, such as linear block codes and convolutional codes, in decoding the corrupted signals with a very low bit error rate in a strong noise environment. The outstanding error correction decoding

ability of turbo codes is achieved at the cost of the increased computational complexity. That is to say, turbo codes require more computation and longer decoding latency than traditional linear and convolutional codes.

In 1948, Shannon proved that every noisy channel has a maximum rate at which information may be transferred through it and that it is possible to design error-correcting codes that approach this capacity, or Shannon limit, provided that the codes may be unbounded in length. For the last six decades, coding theorists have been looking for practical codes capable of closely approaching the Shannon limit.

For more than four decades, *National Aeronautics and Space Administration* (NASA) and the *Jet Propulsion Laboratory* (JPL) have been sending deep-space probes to explore the far reaches of our solar system. Because of the extreme dilution of signal power over interplanetary distances, JPL is always looking for codes that approach Shannon limit as closely as possible. In late 1950s and early 1960s, the data received from probes were uncoded. By the late 1960s and early 1970s, missions were using codes, such as Reed-Muller and long constraint length convolutional codes. In 1977, the Voyager was launched with an optimized convolutional code and a suboptimal Viterbi decoder. This convolutional code was concatenated with a (255,223) Reed-Solomon code. In 1993, turbo codes were proposed by *Berrou, Glavieux and Thitimajshima*. The key insights were the introduction of an interleaver between the two convolutional codes and the iterative suboptimal decoding. Before Cassini launched in 1996, JPL had already begun the standardization of turbo codes for future space missions. In 1998, MacKay visited JPL to present a talk on *low density parity check* (LDPC) codes. It was shown that the LDPC codes introduced from Gallager's thesis [3] in 1963, can be designed to perform as well as, or better than turbo codes.

The history of error correction codes illustrates that turbo codes are one of the most effective channel coding schemes in the world. The renaissance of LDPC codes did not mark the end of turbo codes, however, LDPC codes have performance and complexity advantages over turbo codes at high code rates, but turbo codes are currently still the best

solutions for the lower code rates.

For a long time, the hardware complexity has hampered turbo codes' wide application in communication systems. However, recent developments in *very large scale integrated-circuit* (VLSI) design enables the widespread use of the turbo codes. Now turbo codes are widely applied in communications, especially in *code division multiple access* (CDMA) system, *digital video broadcasting* (DVB) and low space communications. For example, in the third Generation Partnership Project turbo codes have been one of the standard error control coding techniques and an 8-state turbo encoder is applied, as defined in [4]. Another fact is that they are employed in *Consultative Committee for Space Data Systems* (CCSDS). After the first discussions in the May, 1996, at the Meeting of the CCSDS, it was decided, in agreement with NASA and other national space agencies, to include a set of turbo codes in a new issue of the CCSDS telemetry channel coding recommendation. In the future CCSDS applications, turbo codes will be an add-on option to the recommendations without modifying the existing coding schemes and will retain compatibility with the CCSDS Packet Telemetry recommendation.

This thesis is divided into five chapters. The first chapter contains the background introduction. In the second chapter, the main principles of turbo codes are introduced. The structures of both the turbo encoders and the decoders are explained. The core algorithm is discussed. A sliding window technique, which is specially designed for hardware implementation, is introduced. In the third chapter, the decoding performance of turbo codes is investigated in terms of BER. Not only the software simulation results, but also the free distance asymptotic performance of turbo codes are provided. In this chapter, an important issue, the limited number of bits representing a number on hardware implementations, is considered. In the fourth chapter, the hardware implementation of turbo codes is discussed. The design is implemented in both VHDL and FPGA boards. The hardware simulations are compared with the software simulations to verify the decoding correctness. A pipelining technique, which significantly reduces the decoding latency, is introduced in the turbo decoding. In the hardware implementation, the different turbo

decoding schemes are also evaluated by the decoding latency, which is finally represented by the number of clock cycles. The implementation of a turbo decoder extending to general cases is discussed. Finally, some further improvements are discussed. In the last chapter, a summary of the thesis is given.

1.2 Motivation for the Research

First, turbo codes are fascinating because their performance is so close to the Shannon limit. In some best cases for low code rates, they are only 0.7dB away from the Shannon limit, which is not achievable with linear and convolutional codes with such low code rates. We are interested in the excellent performance of turbo codes and the factors that contribute to this performance, even if the structure of turbo encoders are rather simple. We study their architectures, explore the properties, and investigate the performance. We want to know, what are the key factors that make the turbo codes outstanding from other coding schemes?

Second, the turbo decoding is of high computational complexity. The software implementation is actually an idealized model, however, it is the first step to deeply understand the turbo codes. Furthermore, we attempt to develop a turbo decoder in a hardware device, to implement its complex algorithm and iterative decoding. This implementation can be regarded as a real application. It will take us to focus on more considerations of hardware beyond modeling in mathematics. It is critical that we meet the requirements of decoding correctness, decoding latency, and hardware complexity, as well as to look for an effective tradeoff among these requirements.

Third, Xilinx FPGA platforms are popularly employed in hardware design. One of its evaluation platforms in university programs, XUPV5-LX110T FPGA board, seems a good choice to us. We attempt to implement a turbo decoder on this board, make efforts to improve the decoding performance in latency and hardware complexity, and then extend

the original implementation to more general cases.

Chapter 2

Principles of Turbo Codes

In a communication link, the turbo encoder is employed to convert each information bit to a short message, called a symbol. The symbols are transmitted through a channel, either a wired or wireless channel. The symbols are corrupted by noise when they travel through the channel. At the end of the channel, the received signals are fed into the turbo decoder. The turbo decoder implements an optimal or sub-optimal algorithm, to decode the signals in an iterative way. Finally, the receiver makes a hard decision to judge what is transmitted. In this chapter, the architectures and mechanisms of turbo encoders and decoders are introduced. The components are discussed in details to explore the properties that make significant contributions to the decoding performance.

2.1 The Architecture of Turbo Encoders

2.1.1 RSC Encoders and Turbo Encoders

The turbo encoder consists of RSC encoders and interleavers, as shown in Figure 2.1 [5]. This turbo encoder is formed by a parallel concatenation of two constituent RSC encoders separated by an interleaver, and this is referred to as a *parallel concatenated convolutional*

code (PCCC).

In Figure 2.1, a code rate $1/3$ 4-state turbo encoder block diagram is shown. The dashed box represents an RSC encoder, which is developed from the convolutional encoders by feeding back one of its two outputs to the input. Two RSC encoders are connected in parallel with an interleaver in the middle, denoted by **I**. These two constituent RSC encoders are identical. They could, however, be different for more complex turbo codes. For the RSC encoder in the dashed box, the memory order M , which is also called memory length and defined as the number of shift registers denoted by **D**, is 2. The constraint length, $\nu = M + 1$, equals 3. The interleaver **I** is used to permute the source bits denoted by u ; the permuted sequence is applied at the input of the second RSC encoder. For this encoder, every information bit u is encoded to a short message of 3 bits, v_0 , v_1 , and v_2 , which is called a symbol. The code rate is defined as the ratio of the input bits to the output bits. Obviously, the code rate is $1/3$.

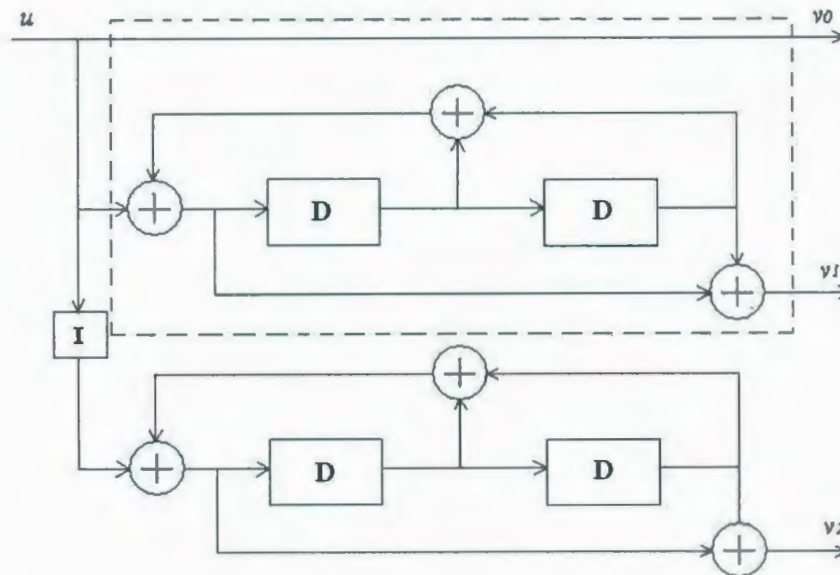


Figure 2.1: A rate $1/3$ turbo encoder

The information bit u is also considered as a bit sequence, because there is always more than one information bit and the information bits arrive successively. So are the output bits v_0 , v_1 , and v_2 . It can be seen that the every output sequence is a linear combination

of the information sequence. The relationship between the information sequence u and the output sequences $v_0v_1v_2$ can be described by the generator matrix. In a turbo encoder, the two constituent RSC encoders are generally same. Therefore, a turbo encoder is described by either of the RSC encoders, which is expressed by a generator matrix. The generator matrix can be represented as $G(D) = [1 \quad g_1(D)/g_0(D)]$, with $g_0(D)$ and $g_1(D)$ as the feedback and the forward polynomials, respectively. These polynomials can be represented in a binary or octal format. For example, for the encoder in Figure 2.1, $g_0(D) = 1 + D + D^2$, can be denoted in octal by 7 (111), and $g_1(D) = 1 + D^2$, can also be denoted by 5 (101). So, for the rate $\frac{1}{2}$ component RSC code in this turbo encoder, the generator matrix is $G(D) = [1 \quad (1 + D^2)/(1 + D + D^2)]$.

RSC encoders are derived from convolutional encoders. Some properties of RSC encoders are inherited from the parent convolutional encoders. For example, the codeword weight distribution and spectrum. An RSC encoder, denoted by $G_1(D) = [1 \quad g_1(D)/g_0(D)]$, can be derived from its original convolutional encoder, denoted by $G_0(D) = [g_0(D) \quad g_1(D)]$, by feeding back its output $g_0(D)$ to its input. Any information sequence which finally terminates this RSC encoder into all-zero state S_0 can be represented by $p(D)g_0(D)$, with $p(D)$ as a polynomial. This information sequence generates the codeword $[p(D)g_0(D) \quad p(D)g_1(D)]$ by this RSC encoder. Obviously this codeword can also be generated by the sequence $p(D)$ when the original convolutional encoder $G_0(D) = [g_0(D) \quad g_1(D)]$ is used. Therefore, all codewords $p(D)g_0(D)$ and $p(D)g_1(D)$ generated by an RSC encoder with the input $p(D)g_0(D)$ can be obtained by the corresponding convolutional encoder with the input $p(D)$, and vice versa. Consequently, the RSC encoder and its original convolutional encoder have a common codeword weight distribution and spectrum [5].

Now an example is used to illustrate the relationship between convolutional encoders and RSC encoders. The RSC encoder $G_1(D) = [1 \quad (1 + D^2)/(1 + D + D^2)]$ in Figure 2.1 can be derived from the convolutional encoder $G_0(D) = [(1 + D + D^2) \quad (1 + D^2)]$. Given $p(D) = 1$, $p(D)$ followed by two '0' terminates the convolutional encoder $G_0(D)$ into all-zero state. The two output sequences are "111" and "101", respectively. On the other

hand, $p(D)g_0(D)$ is equal to $g_0(D)$, the sequence “111”. If this sequence is fed into the RSC encoder $G_1(D)$, the RSC encoder is also terminated into all-zero state and the output sequences are “111” and “101”, the same as what we got from the convolutional encoder $G_0(D)$. Therefore, we always get same output sequences whether $p(D)$ is fed into the convolutional encoder $G_0(D)$ or $p(D)g_0(D)$ is fed into the RSC encoder $G_1(D)$. Therefore, this convolutional encoder and the derived RSC encoder have the same codeword weight distribution.

Another component in a turbo encoder is the interleaver. The interleaver is viewed as a function of permutation. If a sequence is interleaved, a new sequence is obtained. For example, if “1234” is interleaved, then “2314” may be generated. In this new sequence, all elements can be found in the original sequence, but they are in the different positions. The discussion of the interleaver can be found in the section of turbo decoder architecture.

2.1.2 The AWGN Channel Simulation

The Gaussian random process plays an important role in communication systems. The fundamental reason for its importance is that thermal noise in electronics devices, which is produced by random movement of the electrons due to thermal agitation, can be closely modeled by a Gaussian random process. The reason for the Gaussian behavior of the thermal noise is due to the fact that the current introduced by the movement of electrons in an circuit can be regarded as the sum of the small currents of a large number of sources, namely individual electrons. It can be assumed that at least a majority of these sources behave independently. By applying the central limit theorem, this total current has a Gaussian distribution [6].

The channel model in our design is the *additive white Gaussian noise* (AWGN) model. In this channel, the noise is regarded as the white Gaussian noise, which means the same noise power in all frequencies. In order to simulate this channel, we start from a zero-mean Gaussian noise variable. A zero-mean Gaussian noise variable, denoted as n , can be

generated by using:

$$n = h \cos(2\pi \cdot u_2), \quad (2.1)$$

$$h = \sigma \sqrt{2 \ln \frac{1}{1 - u_1}}, \quad (2.2)$$

where u_1 and u_2 are uniformly distribution variables between 0 and 1, σ is the standard deviation for this Gaussian variable, and h is an intermediate variable. This method is called Box-Muller method. For every bit v traveling through the channel, a noise n is generated and added to the bit. In the end of the channel, signal r is received as $r = v + n$. Figure 2.2 is used to verify the noise simulation by the Box-Muller method. The left histogram illustrates the statistic results by the Box-Muller method, while the right shows the noise histogram by the standard MATLAB function. The number of noise samples is 100,000. It can be seen that the noise distributions in two histograms are close to each other. Both histograms have a shape of Gaussian distribution. Therefore, it can be inferred that Box-Muller method employed in channel simulations is reasonable and acceptable.

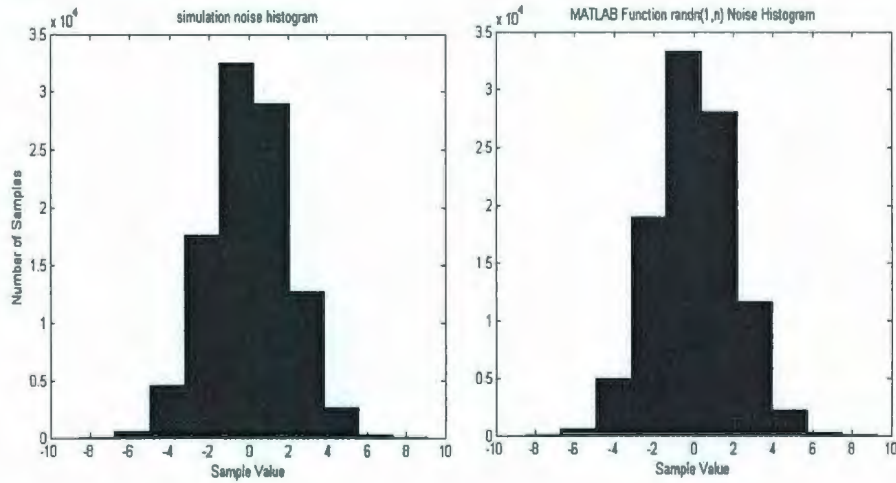


Figure 2.2: Noise histogram comparison

We consider the noise as the one-sided AWGN with the *power spectrum density* (PSD) equal to N_o . If the noise is AWGN with zero mean and one-sided PSD N_o , then this is modeled as a Gaussian random variable with zero mean and variance $\frac{N_o}{2}$ [2]. Decoding algorithms employ the *symbol-energy to noise-spectral-density ratio*, denoted by $\frac{E_s}{N_o}$, while

the performance curves provide the error probabilities depending on the *bit-energy to noise-spectral-density ratio*, denoted by $\frac{Eb}{No}$, with Es as the symbol energy and Eb as the bit energy. For a turbo code, $\frac{Es}{No} = \frac{Eb}{No} \times R$, with R as the code rate. In general, $\frac{Es}{No}$ and $\frac{Eb}{No}$ are expressed in decibels. When the rate is 1/3, $\text{dB}(\frac{Eb}{No}) = \text{dB}(\frac{Es}{No}) + 4.771$. For a punctured code with a rate of 1/2, $\text{dB}(\frac{Eb}{No}) = \text{dB}(\frac{Es}{No}) + 3.01$.

2.2 The Architecture of Turbo Decoders

In the classical architecture of a turbo decoder, there are two blocks of component decoders called *Soft-In and Soft-Out* (SISO) units [2]. The decoding process is iterative, as reliability information for the decoded bit '0' or '1' is exchanged between the two SISO units. Here the reliability information is a value giving a hint of bit '0' or '1'. A large positive value denotes a large probability of bit '1'. A large negative value denotes a large probability of bit '0'. The iterative process stops after several iterations, and then the decoded sequence can be generated by the output(s) of both/either SISO in the last iteration. In this section, both the iterative decoding process and the *Maximum A Posteriori Probability* (MAP) algorithm are presented.

2.2.1 The MAP Algorithm

In turbo codes, several trellis based decoding schemes can be used as core algorithms in SISO units in various applications, such as uni-directional *soft output Viterbi algorithm* (SOVA), bi-directional SOVA, and MAP algorithms. Compared with the SOVA algorithm, the MAP algorithm achieves better performance at the cost of higher computational complexity [2].

In our research, we study the MAP algorithm, which is also called the BCJR algorithm, named after its founders *Bahl, Cocke, Jelinek, and Raviv* who discovered it in 1974. The MAP algorithm minimizes the BER instead of the *word error rate* (WER) [2]. Minimizing

the BER is practically equivalent to minimizing the probability of $P(\hat{u}_\ell \neq u_\ell | \mathbf{r})$, which is the probability, based on the received sequence \mathbf{r} , that the decoded bit of \hat{u}_ℓ is not equal to the transmitted information bit u_ℓ at bit ℓ , with ℓ as the bit index, $\ell = 0, 1, \dots, k-1$, and k as the sequence size. We also refer to the decoding for the bit ℓ as the decoding at stage ℓ .

In the MAP algorithm, the *A Posteriori Probability Likelihood values* (APP L-values) are used for hard-estimation. To obtain APP L-values, first, the branch metrics for each stage can be calculated by [2]

$$\gamma_\ell(s', s) = p(u_\ell)p(\mathbf{r}_\ell | \mathbf{v}_\ell) = p(u_\ell) \left(\sqrt{\frac{Es}{\pi \cdot N_o}} \right)^t \exp \left\{ -\frac{Es}{N_o} \|\mathbf{r}_\ell - \mathbf{v}_\ell\|^2 \right\}, \quad (2.3)$$

where ℓ is the stage number, \mathbf{r}_ℓ is the vector of received values at stage ℓ , \mathbf{v}_ℓ is the corresponding codeword transmitted at stage ℓ , s' is the encoder state at stage ℓ , s is the encoder state at stage $\ell+1$, t is the length of the vector \mathbf{r}_ℓ and \mathbf{v}_ℓ , and $p()$ is the probability density function. This equation can be viewed as computing a probability of a Gaussian variable with multiple dimensions, while in each dimension the noise conforms to an independent and identical Gaussian distribution. From this equation it can be seen that the MAP algorithm is of high complexity because of exponential operations. However, the computational complexity of $\gamma_\ell(s', s)$ can be reduced in the log-domain, thus (2.3) becomes [2],

$$\gamma_\ell(s', s) = \frac{1}{2} u_\ell L_a(u_\ell) + \frac{L_c}{2} (\mathbf{r}_\ell \bullet \mathbf{v}_\ell), \quad (2.4)$$

where $L_a(u_\ell)$ is the *a priori probability* of a information bit u_ℓ , $L_c = \frac{4Es}{N_o}$ is the channel reliability factor influenced by the *symbol-energy to noise-spectral-density ratio* in the channel, and “ \bullet ” denotes the inner product of two vectors. If *binary phase shift keying* (BPSK) modulation is applied, the information bit u_ℓ is always +1 or -1. If the information bits $u_\ell = -1$ or $u_\ell = +1$ are of equal probability, then the a priori probability $L_a(u_\ell)$ is 0. The reliability L_c is regarded as a constant due to the channel property. The vector \mathbf{v}_ℓ is

actually the codeword, which can be read from the state or trellis diagram of the encoder, corresponding to the information bit u_i . So in the right side of this equation, only received signals r_ℓ are affected by the signal transmission through the channel, while other variables are pre-determined. For each stage, there are $2^{(M+1)}$ branches, with M as the memory length of the turbo encoder. The calculation of branch metrics should be completed before starting to calculate other metrics.

Second, we calculate the forward metrics $\alpha_\ell^*(s')$ and the backward metrics $\beta_\ell^*(s')$ by using the branch metrics $\gamma_\ell(s', s)$. The calculation for both metrics is similar. The only difference between them is that the forward metrics are calculated from the first stage of the block, while the calculating of backward metrics starts from the last stage. The forward metrics $\alpha_\ell^*(s')$ and the backward metrics $\beta_\ell^*(s')$ are obtained by using [2]

$$\alpha_{\ell+1}^*(s) = \max_{s' \in \sigma_\ell} [\gamma_\ell^*(s', s) + \alpha_\ell^*(s')], \quad (2.5)$$

and

$$\beta_\ell^*(s') = \max_{s \in \sigma_{\ell+1}} [\gamma_\ell^*(s', s) + \beta_{\ell+1}^*(s)], \quad (2.6)$$

where $s' \in \sigma_\ell$ denotes the branch set at stage ℓ , in which any branch has a starting state of s' . Note that there is an approximation applied to these equations [2]. It now becomes

$$\ln(e^{x_1} + e^{x_2}) = \max^*(x_1, x_2) = \max(x_1, x_2) + \ln(1 + e^{-|x_1 - x_2|}). \quad (2.7)$$

In (2.7), it can be seen that the computational complexity is still high because of the exponential operations on the right side. Further approximation can be applied to simplify the computation by replacing the second item on the rightmost side. For example, using a linear expression in segments to replace $\ln(\cdot)$. However, the simplest way is to ignore the item $\ln(\cdot)$ as follows:

$$\ln(e^{x_1} + e^{x_2}) = \max^*(x_1, x_2) \approx \max(x_1, x_2). \quad (2.8)$$

Although this simplification brings some computational errors, it significantly reduces the computational complexity. With this simplification, the calculations of forward and backward metrics include only additions and comparisons, which are convenient especially in the hardware implementation. If we employ Equation 2.7 in all $\max^*(\cdot)$ functions, the algorithm is called the LOG-MAP algorithm. If we further employ Equation 2.8, it is called the MAX-LOG-MAP algorithm. For each stage, there are 2^M forward and backward metrics, respectively.

When the difference between x_1 and x_2 is 0.1, $\ln(1 + e^{-|x_1 - x_2|}) = 0.64$. In this case, computational errors may occur. When the difference between x_1 and x_2 is larger than 6, the item $\ln(1 + e^{-|x_1 - x_2|})$ is less than 0.0024756. In practice cases, the difference between metrics is mostly in magnitude of 10^{-1} at the very beginning, and becomes larger in the iterative decoding. Therefore, the approximation in Equation 2.8 brings the decoding performance different. There is a coding gain of $0.3 \sim 0.5\text{dB}$ if the LOG-MAP algorithm is employed instead of the MAX-LOG-MAP. However, when the SNR is larger than 1.5dB, simulations show that the LOG-MAP and MAX-LOG-MAP algorithms generally make no difference in decoding performance in terms of BER.

Finally, we can obtain the APP L-value and/or extrinsic probability by using the metrics $\alpha_\ell^*(s')$ in (2.5), $\beta_\ell^*(s')$ in (2.6) and $\gamma_\ell^*(s', s)$ in (2.4).

$$L(u_\ell) = \max_{(s', s) \in \sum_\ell^+} [\alpha_\ell^*(s') + \gamma_\ell^*(s', s) + \beta_{\ell+1}^*(s)] - \max_{(s', s) \in \sum_\ell^-} [\alpha_\ell^*(s') + \gamma_\ell^*(s', s) + \beta_{\ell+1}^*(s)], \quad (2.9)$$

where \sum_ℓ^+ and \sum_ℓ^- denote the positive branch set and the negative branch set in stage ℓ , respectively. Since the computation is in the *log* domain, the APP L-values are called *log likelihood ratio* (LLR) values as well. For each stage, there is only one APP L-value used as reliability information. From (2.9) it can be seen that the requirement to calculate the APP L-value is the availabilities of the branch, forward, and backward metrics.

2.2.2 Iterative Decoding

Figure 2.3 shows the iterative turbo decoder architecture based on the MAP, the LOG-MAP, the MAX-LOG-MAP algorithms, with **I** as the interleavers, **DI** as the corresponding deinterleavers, $L_a(u)$ as a posteriori L-values, $L_e(u)$ as the extrinsic probability and **d** as the decoded sequence [5]. The main components of the decoder are interleavers, deinterleavers and two SISO units. In each SISO unit, the MAP algorithm or its simplifications previously introduced is employed. The interleavers have the function of permutation, and the deinterleavers have the inverse permutation functions.

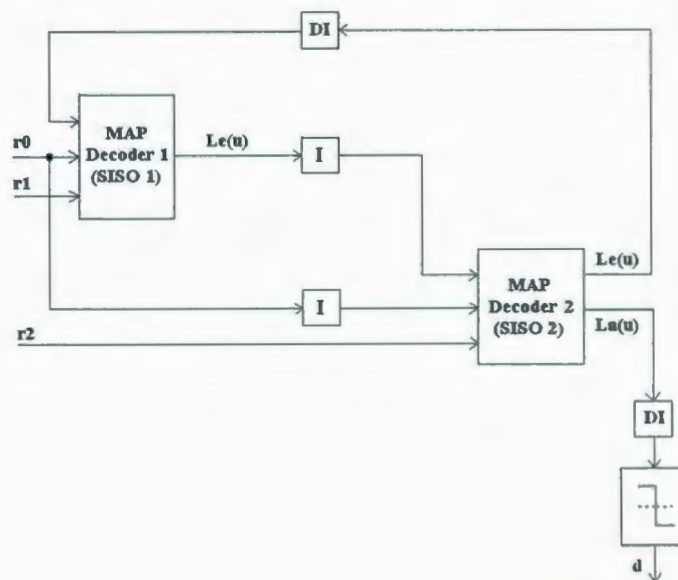


Figure 2.3: An iterative turbo decoder with the MAP algorithm

In the iterative decoding, $L_e(u)$ values, which are referred to as the *extrinsic probabilities*, from the first component decoder are used as the a priori probabilities $L_a(u)$ for the second decoder when computing the branch metrics, and $L_e(u)$ values from the second decoder are used as $L_a(u)$ for the first decoder in the next iteration. The extrinsic probabilities $L_e(u)$ do not depend on the information bits u_ℓ and the a priori probabilities $L_a(u)$, and results from a posteriori L-values $L_a(u)$, by subtracting all items that are related to

the information bits u and the a priori probability $L_a(u)$. From (2.4),

$$\gamma^*(s', s) = \frac{1}{2}u_\ell \cdot L_a(u_\ell) + \frac{L_c}{2}u_\ell \cdot r_{u_\ell} + \frac{L_c}{2}p_\ell \cdot r_{p_\ell}, \quad (2.10)$$

where u_ℓ is the information bit and p_ℓ is the parity bit at stage ℓ , and r_{u_ℓ} and r_{p_ℓ} are the received signals corresponding to these bits [2]. In this equation, the first and second terms on the right side, which contain u_ℓ and/or $L_a(u_\ell)$, should be excluded when computing the extrinsic probability. Therefore,

$$L_e(u_\ell) = \max_{(s', s) \in \sum_\ell^+} [\alpha_\ell^*(s') + \frac{L_c}{2} \cdot p_\ell \cdot r_{p_\ell} + \beta_{\ell+1}^*(s)] - \max_{(s', s) \in \sum_\ell^-} [\alpha_\ell^*(s') + \frac{L_c}{2} \cdot p_\ell \cdot r_{p_\ell} + \beta_{\ell+1}^*(s)], \quad (2.11)$$

where $\alpha_\ell^*(s')$ and $\beta_{\ell+1}^*(s)$ are the forward metric at stage ℓ and backward metric at stage $\ell + 1$, respectively [2].

In general, the information bits of “0” or “1” are symmetric, e.g., the transmitter sends bit “0” and “1” with an equal probability. Therefore, in the beginning of the iterative decoding, the $L_a(u_\ell)$ that are used to calculate the branch metrics in Equation 2.4 are initialized to 0. In the iterative decoding, the extrinsic probabilities $L_e(u)$ are calculated instead of the APP L-values $L(u)$ in the MAP/LOG-MAP/MAX-LOG-MAP algorithm. The extrinsic probabilities $L_e(u)$ from one SISO unit are always used as the a priori probabilities $L_a(u)$ to calculate the branch metrics in the next SISO unit. After several iterations, the decoding process stops. In the last half iteration, the APP L-values in Equation 2.9 are released instead of the extrinsic probabilities $L_e(u)$. These APP L-values are finally applied to determine the hard estimation. If the sign of an APP L-value is positive, then this bit is decoded as “1”, else it is decoded as “0”.

2.2.3 Interleaver Design

Interleaving is a practical technique to enhance the error correcting capability of iterative coding. The main function of the interleaver is to deliberately permute the sequence of bits in a block, so that successive errors in communications can be distributed over a wide range by an interleaver. Inserting an interleaver between the channel encoder and the channel is an effective method to cope with burst errors [7].

There are four main types of interleavers: block interleavers, convolutional interleavers, random interleavers and code matched interleavers. The simplest are the block interleavers. Figure 2.4 shows the structure of block interleavers. A block interleaver formats the input sequence in a matrix of m rows and n columns, a size of $m \times n$. The input sequence is written row by row and read out column by column. Its function is:

$$\pi(i) = [(i - 1) \bmod n] \times m + \lfloor (i - 1)/n \rfloor + 1, \quad i \in A. \quad (2.12)$$

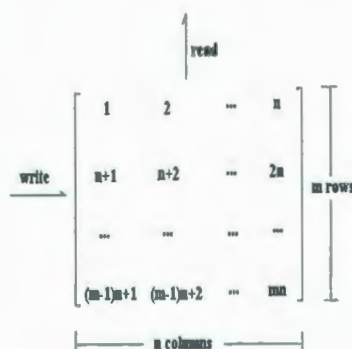


Figure 2.4: The structure of a block interleaver

The introduction of convolutional interleavers and code matched interleavers can be found in [5]. Another frequently used interleaver in turbo codes is the random interleaver. In general, the random interleavers outperform the block interleaver due to their constraints which helps to eliminate the burst errors.

For example, the spread-random (S-random) interleaver, proposed by *Divsalar and*

Pollara in [8], is a complex interleaver. The S-random interleavers are based on the random generation of k integers from 1 to k with an S-constraint, where S is defined as the minimum interleaving distance. Some S-random interleaver is characterized by two parameters S_1 and S_2 , with S_1 as the spread and S_2 as the minimum interleaving distance. Each randomly selected integer is compared with S_1 previously selected integers. If the absolute value of the difference between the currently selected integer and any of the S_1 previous selected integers is smaller than S_2 , then the currently selected integer is rejected, or it is accepted. It can be described as

$$|i_1 - i_2| \geq S_2, \quad (2.13)$$

whenever

$$|\pi(i_1) - \pi(i_2)| \leq S_1, \quad i_1, i_2 \in A, \quad (2.14)$$

where i_1 and i_2 are the position indexes, A is an index set containing from 1 to k , and $\pi(i)$ is the interleaver function.

In the algorithm to construct such an S-random interleaver, there is no guarantee that the process will finish successfully [9] due to the randomness. Furthermore, the search time for the algorithm become prohibitively large for large S_1 and S_2 . A good trade-off between interleaver performance and search time is obtained for $S_1, S_2 < \sqrt{\frac{k}{2}}$, with k as the interleaver size.

Another random interleaver is specified in [10]. The algorithm for the interleavers, which can be also called permutators, is described in detail for some different block length k . Assuming the indices of bits in the block are denoted by the integers $1, 2, \dots, k$, the size k is expressed by $k = k_1 k_2$. The parameters k_1 and k_2 are given in Figure 2.5 for a few specified block size [10].

Next, do the following operations in the loop for $s = 1$ to k to obtain the permutation numbers. In the equation below, $\lfloor x \rfloor$ denotes the largest integer less than or equal to x , and p_q is defined as following:

$$p_1 = 31; p_2 = 37; p_3 = 43; p_4 = 47; p_5 = 53; p_6 = 59; p_7 = 61; p_8 = 67$$

Information block length	k_1	k_2
1784	8	223
3568	8	223×2
7136	8	223×4
8920	8	223×5
16384	(note)	(note)
NOTE – These parameters are currently under study and will be incorporated in a later revision.		

Figure 2.5: Parameters k_1 and k_2 for the specified information block lengths

$$\begin{aligned}
m &= (s - 1) \bmod 2 \\
i &= \lfloor \frac{s - 1}{2k_2} \rfloor \\
j &= \lfloor \frac{s - 1}{2} \rfloor - ik_2 \\
t &= (19i + 1) \bmod \frac{k_1}{2} \\
q &= t \bmod 8 + 1 \\
c &= (p_q j + 21m) \bmod k_2 \\
\pi(s) &= 2(t + c \frac{k_1}{2} + 1) - m
\end{aligned} \tag{2.15}$$

where m, i, j, t, q , and c are internal variables, s is the original index of the bits in the block and $\pi(s)$ is the index of the permuted bits. Besides those types of interleavers, there are some more complex and effective interleavers, for example, “dither” interleavers in [11]. It is found that interleavers influence significantly the turbo code decoding performance at BER. The answer to the observation is that the codeword weight spectrum is changed after the information bits are interleaved. It is the codeword weight spectrum that determines the performance bounds at BER. The spectrum of the codewords is referred to as the Hamming weight distribution of all codewords except the all-zero codeword. Research shows that by using pseudo-random interleavers the spectrum of the codewords is thinned [2]. This phenomenon is called “spectrum thinning effect” that refers to the concentration

of codeword weight spectrum distribution after the information bits are interleaved. In spectrum thinning effect, the number of codewords of moderate Hamming weights increases, while the number of codewords of low and high weights decreases. Such a "thinning effect" results in a reduced bit error probability. As in a convolutional code the decoding error rate is bounded by the codeword distance spectrum and the basic component RSC encoder of a turbo code is developed from the convolutional codes, it is not difficult to understand that the turbo code performance is influenced by the change of codeword distance spectrum, which is caused by the interleavers.

A well-designed interleaver, e.g., a pseudo-random interleaver, can significantly improve the decoding performance. Therefore, the design of an effective interleaver is one of the most important ways to improve the decoding performance. In the turbo decoder, the architecture of deinterleaver is fully determined by the function of the interleaver. As a result of the interleaving/deinterleaving operations, burst errors are spread out in time so that the errors in various positions are independently distributed. After selecting an interleaver type, the size of the interleaver is the most significant parameter we need to consider. In general, a larger size of interleaver often provides a better decoding performance. In real applications, the interleaver size is much larger than the encoder memory length M . For example, in the 4-state encoder the memory length is 2, while the interleaver size can be 128, 512, 1024, 4096, 65536, or even larger.

The selection of an interleaver is a critical issue in the turbo code design. To find out a turbo code with superior performance, the design may be extensively investigated in the software simulations. However, the architecture of the interleaver has already been determined schematically before the hardware implementation. Whatever an interleaver is, it is viewed as a mapping function to permute data in a long sequence. It can be implemented by a look-up table. This look-up table may only provide mapping addresses for a memory block shared by an interleaver and its corresponding deinterleaver. The size of this look-up table generally does not depend on the type of the interleaver. In hardware implementation, we are not seeking for a turbo code with the best performance in terms of

BER, but implementing a fully investigated design. However, the look-up tables for both the interleavers and the Trellis of the constituent RSC encoders are replaceable. Thus, the hardware implementation is more flexible. For simplicity, block interleavers are employed. If the block size is an even number, a look-up table is omitted. The address mapping for the interleave/deinterleaver is realized by only changing the low bits with the high bits in addresses. For example, if an block interleaver 8×8 is considered, the data addresses may be represented in 6 bits, $b_5b_4b_3b_2b_1b_0$. All data can be simply relocated in the deinterleavers as $b_2b_1b_0b_5b_4b_3$.

2.2.4 Sliding Window Technique

In the introduction of the MAP algorithm, it can be seen that the calculation of forward metrics starts from the first stage of a block, while the calculation of backward metrics starts from the last stage of a block. It means the minimum length of decoding is the length of a block when applying the MAP algorithm. In general, this length is the size of the interleaver/deinterleaver. In Figure 2.3, it can be seen that a full block has been decoded by the MAP algorithm in one SISO unit before it is delivered to the next SISO unit. Therefore, in the iterative decoding process, the minimum requirement to decode the signals successively is the requirement to decoding a block of interleaver/deinterleaver size.

Sliding window techniques were discussed in [12], [13], [14], and [15], especially when hardware implementation is considered. In general, the size of the interleaver is chosen to be large so that a desirable decoding performance can be achieved. However, the larger the block is, the more memory we need to store the metrics. A large memory always denotes a high hardware complexity. Generally, a large size of block, for example, 1024 symbols, makes the design infeasible in hardware implementation because of the extreme long decoding delay and large memory to store metrics of 1024 stages. In this case, the minimum decoding delay for one bit is the decoding time used for a block. The first bit is released until the entire block is calculated, due to the backward computation starting at

the last bit in the block. For a turbo code with memory length of 4, there are 32 branch, 16 forward, and 16 backward metrics in each stage. If each metric occupies 16 bits, then the minimum memory for the branch and backward metrics are $1024 \times (16 + 16 + 32) \times 16 = 1,048,576\text{bits}$.

The sliding window technology, as shown in Figure 2.6, is introduced to overcome the disadvantages of a long decoding delay and a high hardware complexity. It is suggested that the entire symbol block can be divided into many small windows. Two successive windows are overlapped. For a window, only the bits corresponding to non-overlapped symbols in this window are released, and the overlapped symbols in this window enters into the next window. Each time only one window is loaded and decoded. So the decoded window passes across the block. Since in each window the backward metrics are always re-calculated from the last bit, the cost of the sliding window technique is the recalculation of backward metrics in the overlapped area in each window. For a block, the length of overlapped area is close to the size of the block.

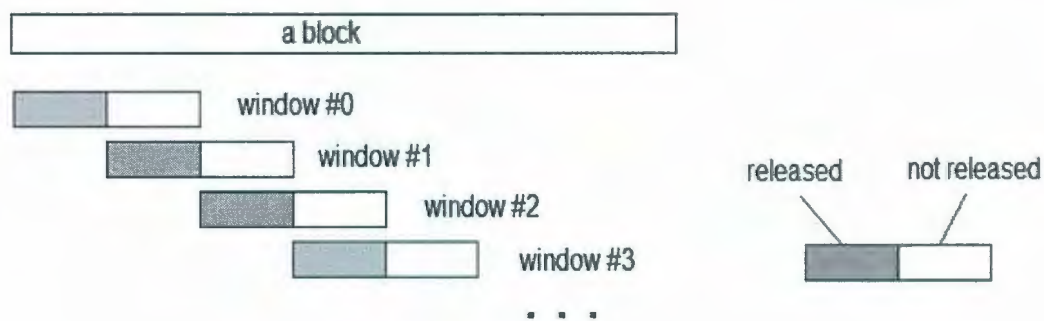


Figure 2.6: A sliding window technique

In general, the size of the window is at least 5 times of constraint length of the encoder [5], which is far smaller than the size of the block, so that the hardware area of the decoding algorithm is significantly reduced, and the decoding delay is reduced to the delay for decoding a window, not a block. The decoding starts when the symbols of the first window are received. Therefore, the decoding is implementable in hardware. However, there is a trade-off. Because the computation of the backward metrics does not start from

the last stage of the block, but the last stage of a window, the backward metrics are not perfect and the decoding is not optimal. Thus, there is some decoding degradation when sliding window technique is applied.

2.3 Performance Analysis for Turbo Codes in the AWGN Channel

The structure of the RSC encoders and the interleavers in the turbo encoders, determine the decoding performance bound in terms of BER. The decoding performance of a turbo code is upper bounded. In the AWGN channel, there is an error-floor region that ranges from moderate to high SNRs, generally starting from $E_b/N_0 = 2$ dB. In that region, the decoding performance curve of a turbo code becomes flatter, and can be bounded by its free distance asymptotic performance. This free distance asymptotic performance provides an effective estimation of decoding performance in moderate and high SNRs.

2.3.1 Distance Properties of Codewords

Some of the basic concepts of coding are briefly explained in this section. They are essential and critical in deriving the decoding performance bounds of convolutional and turbo codes. These concepts are the *column distance function* (CDF), the *minimum distance*, and the *minimum free distance*.

The *minimum free distance*, denoted by d_{free} , is the minimum Hamming distance between all pairs of complete codewords. Note that a complete codeword is defined as a codeword diverging from state S_0 and merging at S_0 exactly once. If the code \mathbf{C} is linear, then the *minimum free distance* equals the minimum Hamming weight among all complete

codewords, excluding the all-zero codeword. That is to say,

$$\begin{aligned} d_{\text{free}} &\stackrel{\text{def}}{=} \min\{d(y^1, y^2) | y^1 \neq y^2\} \\ &= \min\{w(y) | y \neq 0\}, \end{aligned} \quad (2.16)$$

where $d(y^1, y^2)$ is the Hamming distance between the codeword pair y^1 and y^2 , and $w(y)$ is the Hamming weight of codeword y . For a non-catastrophic codes, the CDF d_i approaches the *minimum free distance* as the length of sequence, denoted by i , increases [16],

$$\lim_{i \rightarrow \infty} d_i = d_{\text{free}}. \quad (2.17)$$

Note that a catastrophic code is a code, whose corresponding state diagram contains a circuit in which a nonzero input sequence corresponds to an all-zero output sequence.

There are some other frequently used concepts, such as the *weight enumerating function* (WEF), the *input-output weight enumerating function* (IOWEF), and the *conditional weight enumerating function* (CWEF). The WEF provides a complete description of the weight distribution of all nonzero complete codewords that diverge from and merge with state S_0 exactly once. In general, the WEF is expressed as $A(X) = \sum_{d=d_{\text{free}}}^{\infty} A_d X^d$, with d_{free} as the minimum free distance and d as the codeword Hamming weight. For example, the WEF $A(X) = X^3 + 4X^4 + 7X^5$ denotes that there are one codeword of weight 3, four codewords of weight 4, and seven codewords of weight 5. One way to get the WEF is to study the state diagram by finding all the paths that start from S_0 and end at S_0 exactly once. Another straightforward method to get WEF is to exhaustively count the Hamming weights of all possible codewords, especially for some very simple coding scheme with limited codewords. The evaluation of WEF for the entire turbo encoder could be used to determine the performance bounds.

The CWEF and the IOWEF can be derived from the WEF. The IOWEF $A(W, X)$ contains only information about the input and the output weights of each codeword. It is

expressed by

$$A(W, X) = \sum_{w,d} A_{w,d} W^w X^d, \quad (2.18)$$

where $A_{w,d}$ represents the number of codewords with weight d and information weight w . The relationship between WEF and IOWEF is $A(X) = A(W, X)|_{W=1}$. The bit WEF $B(X)$ can be calculated directly from the IOWEF $A(W, X)$ by [2]

$$B(X) = \frac{1}{k} \frac{\partial A(W, X)}{\partial W} \Big|_{W=1}, \quad (2.19)$$

where k denotes the number of information bits.

2.3.2 Performance Bounds for Convolutional Codes

In general, the constituent encoders of a turbo encoder are derived from the convolutional encoders. For example, in the turbo encoder proposed in [1], *RSC* encoders are used as constituent encoders. The *RSC* encoder is constructed from its original convolutional encoder by feeding back one of its outputs to its inputs. Thus, the theories in convolutional codes are the basic tools applied in the analysis of the *RSC* encoders, and the investigation of the performance bounds of convolutional codes is the basis for further research in the performance bounds for turbo codes.

The decoding performance of convolutional codes could be evaluated by a union bound $P_f(E)$, which is described as the sum of the error probabilities of all possible *first-error event* paths. Note that a first error event is made at an arbitrary time unit denoted by t , if the all-zero path (the correct path) is eliminated for the first time at time unit t in favor of a competitor (the incorrect path). The word error probability caused by first error events $P_f(E)$ is [2]

$$P_f(E) < \sum_{d=d_{\text{free}}}^{\infty} A_d P_d, \quad (2.20)$$

where A_d is the number of codewords of weight d . That is, the coefficient of the weight-

d term in the codeword WEF of the code, and P_d is the word error probability of the first error events of weight d . For a *binary symmetric channel* (BSC), the bound can be simplified when d is odd. The probability of the first error event P_d becomes [2]

$$\begin{aligned} P_d &= \sum_{e=\frac{d+1}{2}}^d \binom{d}{e} p^e (1-p)^{d-e} \\ &< 2^d p^{d/2} (1-p)^{d/2}, \end{aligned} \quad (2.21)$$

where p is the bit error probability in the BSC channel. For any convolutional code with codeword WEF $A(X) = \sum_{d=d_{\text{free}}}^{\infty} A_d X^d$, the upper bound can be expressed by

$$P_f(E) < A(X)|_{X=2\sqrt{p(1-p)}}. \quad (2.22)$$

For a small p , the event error probability bound can be dominated by its first term, that is, the free distance term,

$$P_f(E) \approx A_{d_{\text{free}}} [2\sqrt{p(1-p)}]^{d_{\text{free}}} \approx A_{d_{\text{free}}} 2^{d_{\text{free}}} p^{d_{\text{free}}/2} \quad (2.23)$$

If the channel is memoryless, and the transmitted codeword is assumed to be $\mathbf{v} = (-1, -1, \dots, -1)$, and a new variable ρ is introduced as $\rho = \sum_{\ell=1}^d r_{\ell}$, then ρ is a sum of d independent Gaussian random variables, each with mean -1 and variance $\frac{N_o}{2E_s}$, with \mathbf{r} as the received symbols; that is, ρ is a Gaussian random variable with mean $-d$ and variance $d\frac{N_o}{2E_s}$ [2]. Thus,

$$P_d = Q\left(\sqrt{\frac{2dRE_b}{N_o}}\right), \quad (2.24)$$

where $Q(x)$ is the complementary error function. Therefore, for a convolutional code, the word error probability is

$$P_f(E) < \sum_{d=d_{\text{free}}}^{\infty} A_d Q\left(\sqrt{\frac{2dRE_b}{N_o}}\right), \quad (2.25)$$

and the bit error probability is

$$P_b(E) < \sum_{d=d_{\text{free}}}^{\infty} B_d Q \left(\sqrt{\frac{2dRE_b}{N_o}} \right), \quad (2.26)$$

where A_d and B_d are the coefficients of WEF and bit WEF, respectively.

2.3.3 Free Distance Asymptotic Performance of Turbo Codes

Note in the last section, the union bound $P_f(E)$ of the convolutional codes is derived from the concepts of the free distance, *first-error event*, a memoryless channel, and the codeword weight spectrum that is represented by WEF. Compared with convolutional codes, the turbo codes have similar concepts for their codewords, such as the free distance, WEF, and so on. Both the convolutional codes and turbo codes can be decoded by Viterbi algorithm and the MAP algorithm. The difference between the turbo codes and convolutional codes is that in a turbo code the free distance is changed and the codeword spectrum is thinned after the information bits are interleaved. The decoding performance of a turbo code with low memory length is comparable to that of a convolutional code with high memory length. However, they have same concepts to derive the union bound $P_f(E)$.

For a turbo code, the Equation 2.25 could be also applied. This union bound is a tight bound, representing the asymptotic performance of turbo codes. In this bound, the first term in the summation is the dominant term in most cases. To get the performance bounds of a turbo code, all one needs is the free distance d_{free} , the codeword WEF, $A(X)$, or the bit WEF, $B(X)$. The bit error probability can be approximated by the first term in the union bound [17].

$$P_b \approx \frac{N_{\text{free}} \tilde{w}_{\text{free}}}{N} Q \left(\sqrt{d_{\text{free}} \frac{2RE_b}{N_o}} \right), \quad (2.27)$$

where P_b denotes the performance bounds of a turbo code, N_{free} denotes the number of information sequences causing free-distance codewords, and \tilde{w}_{free} denotes the average Hamming weight of the information sequences causing the free distance codewords. This

approximation and its associated graph are called *free distance asymptote* of a turbo code [17]. In general, the Hamming weight of the information sequences causing the free distance codewords are self-terminating weight-2 input sequences. The performance of a turbo code is generally dominated by low-weight codewords, most of which are most likely contributed by self-terminating weight-2 sequences.

For values of E_b/N_0 above the SNR at which the code rate R equals the channel cutoff rate R_0 , that is, for

$$\frac{E_b}{N_0} > R_0 = -\frac{1}{R} \ln(2^{1-R} - 1), \quad (2.28)$$

the first few terms of WEF are necessary for an accurate estimation of performance bounds.

When $R = \frac{1}{3}$, this cutoff rate R_0 is about 2.03 dB.

2.4 Summary

In this chapter, some principles of turbo codes are explained. These include the architecture of encoders, the permuting function of interleavers, the MAP algorithm, and the iterative decoding process. Besides these, we also discuss a sliding window technique which is employed to reduce hardware complexity. Then, theoretical analysis of the decoding performance of turbo codes is presented. The relationship between convolutional codes and turbo codes helps to derive the performance bound of turbo codes. Finally, the free distance asymptote of a turbo code is explained.

Chapter 3

Monte Carlo Simulations of Turbo Codes

3.1 Monte Carlo Methods

Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results. Monte Carlo methods are often used in simulating physical and mathematical systems. Because of their reliance on repeated computation of random or pseudo-random numbers, these methods are most suited to calculation by a computer and tend to be used when it is infeasible or impossible to compute an exact result with a deterministic algorithm.

A deterministic algorithm is an algorithm which behaves predictably. Given a particular input, it will always produce the same output, and the underlying machine will always pass through the same sequence of states. Turbo decoding, including both the iterative decoding and the core algorithm, is deterministic. To evaluate the turbo decoding performance, especially in terms of BER, we need to estimate the error rate depending on the various input patterns, not only a particular input. Here, the randomness in the decoding is introduced by the input patterns.

The randomness of the input patterns is unpredictable and uncontrollable, so that the turbo decoding simulations with a particular input pattern are not reliable to be used in

a general case. However, Monte Carlo methods provide us a way to estimate a predictable and reliable result. In general, the application of Monte Carlo methods need a large amount of random numbers, e.g., a large amount of input patterns. In a turbo decoding simulation, we employed a pseudo random generator to produce a large number of information bits. A simulation with a pseudo random generator is always repeatable. As long as a random seed is provided, all information bits are predetermined even though the number of the information bits are huge. Those information bits are used to feed into the turbo encoder. Then the codewords travel through the channel and the received signals are decoded. A reliable simulation result always depends on this randomness and the large number of information bits.

The usage of Monte Carlo methods spurs the development of pseudo random generators, for example, the discovery of Mersenne twister pseudo random generator. The Mersenne twister is a pseudo random number generator developed in 1997 by Makoto Matsumoto and Takuji Nishimura that is based on a matrix linear recurrence over a finite binary field. The Mersenne twister provides for fast generation of very high-quality pseudo random numbers, which is designed specifically to rectify the flaws found in older algorithms. Its name derives from the fact that the period length of the random number is chosen to be a Mersenne prime. In our simulations, the number of information bits is generally 100 million. This number is very small when it is compared with the long period $2^{19937} - 1$ of the Mersenne twister generator, so that the advantage of the Mersenne twister generator is not fully utilized. We tested to use the Mersenne twister pseudo random generator in our model, but makes little improvement when compared with other simple random generators. Note that the Monte Carlo methods are only employed in the software simulations in our research, not the hardware implementation.

3.2 Performance Investigation of Turbo Decoders

3.2.1 Decoding Performance of a 16-state Turbo Code

In 1993, turbo codes were first proposed by Berrou et al. In their paper, an example turbo encoder was illustrated, as shown in Figure 3.1 [1]. This turbo code is widely studied. It is employed as a typical case for us to start the investigation of the turbo codes decoding performance. Software simulations on this turbo code provide us a basic knowledge to the decoding performance. For example, by selecting different sizes of the interleavers and different iteration numbers, the decoding performance varies.

Figure 3.2 shows a decoding performance curve with error bars at each sample. For this turbo code, we mainly examine decoding performance in the range of low SNRs, from 0 dB to 2.5 dB. The decoding performance is evaluated in terms of BER. In the figure, the BER values represents the average BER of a large number of blocks. Note that error bars, which represent the standard deviation, in the *log* domain are asymmetric. The bars of a fixed length in different positions in this figure do not mean the same value. The lower part of an error bar is always longer than the upper part, but both parts represent a half of the standard deviation.

When $E_b/N_0 < 0.5\text{dB}$, the BER is high and the performance curve is almost flat. In this area, the BER has a magnitude of 10^{-1} . Such a high BER makes the decoded sequence full of errors, therefore, this area is not desirable working area. When E_b/N_0 is between 0.5dB and 1dB, there is a waterfall region in which the BER drops sharply. In this waterfall region, the BER is not low enough. Furthermore, it is unsteady for each block. If the block size is 4096, the BER could be 0.1 for one block and 0.001 for another block. This unsteadiness can also be inferred from the length of the error bars. For example, at 0.7dB, the BER is about 0.01, while the standard deviation is about $(0.03 - 0.01) \times 2 = 0.06$. Therefore, this area is not the desirable working area due to its unsteadiness. From 1dB to

2dB, the BER varies from 10^{-4} to 10^{-5} . The decoding performance in this area is steady and low. There are several bit errors in some block, while in the most blocks there is no error bit. That is why the error bars are long. But the standard deviations are not as large as we expected. At 2dB, the BER is about 10^{-5} , while the standard deviation is about $2 \times (10 - 1) \times 10^{-5} = 1.8 \times 10^{-5}$. This is the desirable working area for turbo codes. Other codes, such as linear block codes and convolutional codes, cannot achieve such low BERs in this strong noise environment.

When E_b/N_0 increases beyond 2dB, the performance curve becomes flatter. It is the *error floor* that prevents BER from falling down sharply after the waterfall region. This floor explains the phenomenon of why the performance of turbo codes are poorer than that of convolutional codes in very high SNRs, for example, at 20dB. This error floor can be approximated by the free distance asymptotic performance.

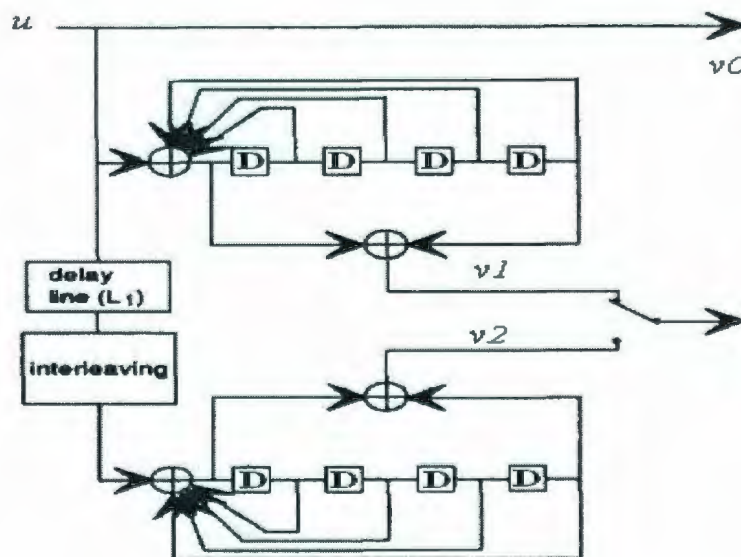


Figure 3.1: A 16-state turbo encoder

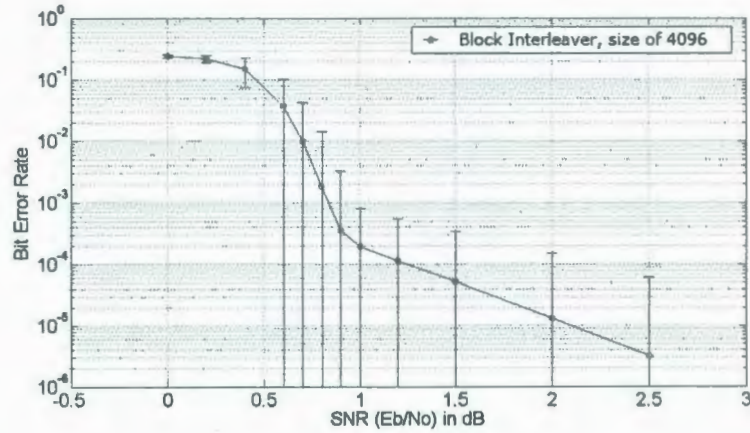


Figure 3.2: Decoding performance of a turbo code

3.2.2 Calculating the Free Distance Asymptotic Performance

The performance bounds in Equations 2.25 and 2.26 represent the asymptotic performance, and the Equation 2.27 is adopted to calculate the free distance asymptote of a turbo code. Five parameters are required to determine the free distance asymptote. These are N , R , d_{free} , N_{free} and \tilde{w}_{free} . N_{free} , also called multiplicity of the information sequences for all possible free distance codewords, is the number of the information sequences for overall free distance codewords. \tilde{w}_{free} is the average Hamming weight of the information sequence corresponding to free distance codewords. To calculate the free distance asymptote, the most difficult task is to obtain the multiplicity N_{free} and the average information sequence weight \tilde{w}_{free} .

The information bit sequence is exclusively composed of zeros and several non-zero bit sequences, if it generates a low weight codeword, because the first constituent encoder is terminated when the information bit sequence is encoded. The number of bit sequences, corresponding to low weight codewords, is assumed to be very small. In general, this number is about 2. Therefore, the Hamming weights of the information and the first parity sequence are small, and not difficult to calculate. However, the Hamming weight of the second parity sequence is more complex to calculate, due to the interleaving function. The

interleaved bit sequence, which is fed into the second constituent encoder, may generate the second parity sequence with a large Hamming weight, because the trellis termination is not guaranteed. The second parity sequence is completely different from the first parity sequence because it represents a completely different path in the trellis diagram. The termination of the second constituent encoder is unknown. Practically, it terminates with a very small probability.

When computing the free distance asymptotic performance, the information sequence is always assumed to be composed of one or two bit sequences corresponding to low weight complete codewords. This assumption is acceptable in most cases, with the reference encoders being good examples. This assumption is made due to the fact that we cannot search all possible combinations of all available bit sequences corresponding to low weight complete codewords. Such a complete search could be prohibitive when the number of bit sequences increases, and/or the information sequence is composed of many bit sequences. If the total number of candidate bit sequences is B and the number of the candidate bit sequences in the information sequence is b , then the maximum searching times is B^b . In addition, each search requires that we exhaustively consider all possible positions for these candidate bit sequences in the information sequence. That is also prohibitive. For example, the information sequence is 4096, while there are 3 different candidate bit sequences and the length of each candidate bit sequence is only 7.

The following explains how to obtain $N_{\text{free}} \times \tilde{w}_{\text{free}}$. First, for the constituent encoder of a turbo encoder, we collect the bit sequences that generate rate 1/2 low weight codewords by this constituent encoder. These bit sequences are viewed as a set of candidate sequences. Second, according to the assumption, we choose one or two sequences in this set. The chosen sequences can be either identical or different. Third, we generate the information sequence by inserting the chosen sequences into an all-zero sequence. The positions of the chosen sequences in the information sequence are arbitrary. Fourth, the information sequence is interleaved, and then fed into the second encoder to generate the second parity sequence. If the Hamming weight sum of information, first parity, and second parity sequences is

equal to the minimum distance, it is added to $N_{\text{free}} \times \tilde{w}_{\text{free}}$. If this sum is larger than the minimum distance, do nothing and skip to the next step. If this sum is smaller than the minimum distance, $N_{\text{free}} \times \tilde{w}_{\text{free}}$ is initialized to this sum. Fifth, we repeat the third and fourth steps to exhaust all positions for the chosen sequences. Finally, we have to repeat the second step to exhaust all bit sequences we get in the first step. A general idea of this algorithm can be seen in [18]. However, we have to modify the algorithm in [18] to ensure the convergence of the program, by setting a maximum length for paths when searching in trellis diagram in the fourth step.

We cannot assert that for turbo codes the information sequence is always composed of only one or two bit sequences corresponding to complete codewords. That means this algorithm might cause computation errors in computing the free distance asymptote. The errors appear because we have not exhausted the case when the information sequence is composed of three bit sequences, or more. In Equation 2.26, the terms in the right hand side should be considered when calculating the performance bounds, if it is quite close to the dominant first term. However, this kind of computation errors are ignored if we only compute the free distance asymptotic performance, not the bounds. In our experience, the information bit sequences, which correspond to free distance codewords, often contains one or two non-zero bit sequences.

3.2.3 Free Distance Asymptotic Performance

In this section, the decoding performance curves of turbo codes are compared with their free distance asymptotic performance. The decoding performance curve is always obtained by software simulations, while the free distance asymptotic performance is estimated by using Equation 2.27. We expect that the free distance asymptotic performance should be close to the decoding performance curve in high SNRs due to the error floor.

Figure 3.3 shows the decoding performance of Berrou's example. In this turbo code, the RSC encoder has the free distance of 6. The rate 1/3 codewords have a free distance of

20. The information bit sequences, which generate codewords of weight 20, are composed of 2 weight-2 bit sequences. A computer search provides the total Hamming weight 16340 in information sequences corresponding to the codewords with an overall Hamming weight 20. That is to say, $N_{\text{free}} \times \tilde{w}_{\text{free}} = 16340$. The free distance asymptotic performance is expressed as:

$$P_b \approx \frac{16304}{4096} Q \left(\sqrt{20 \frac{2E_b}{3 \times N_o}} \right). \quad (3.1)$$

It can be seen that the decoding performance curve is quite close to the free distance asymptotic performance after its waterfall region. For example, at 2.5 dB, the difference between them is only 10^{-6} . That is to say, the decoding performance can be estimated by this free distance asymptotic performance at large SNRs.

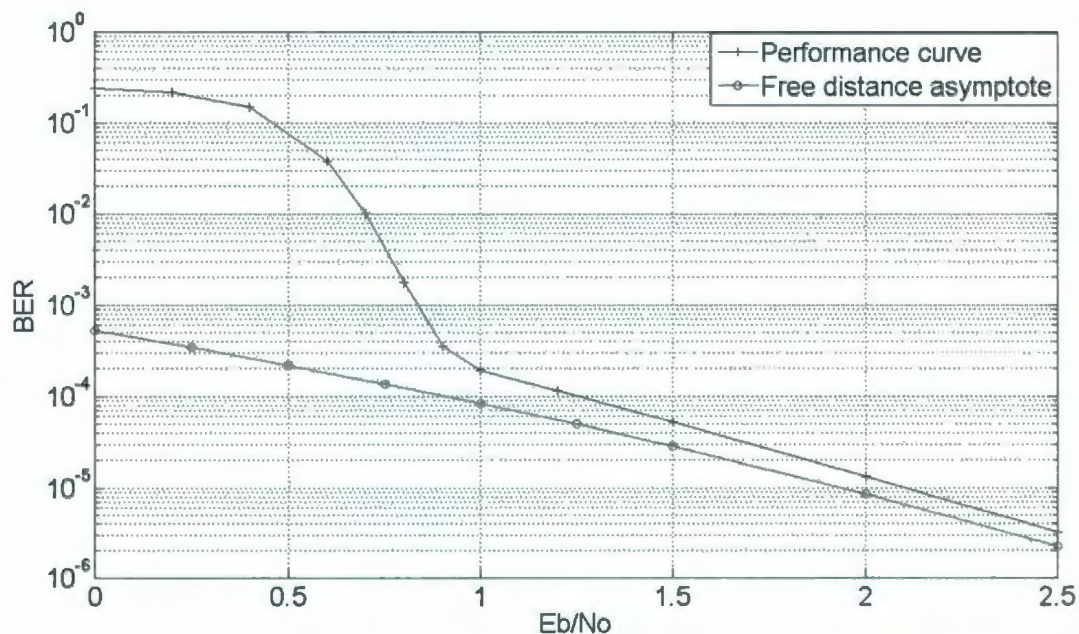


Figure 3.3: Decoding performance and free distance asymptotic performance of the Berrou's example

Figure 3.4 gives another example encoder that is used in [10]. In this turbo code, its RSC encoder has the free distance of 7. The rate 1/3 codewords have a free distance of 24. The information bit sequences, which generate codewords of weight 24, are composed of 2 weight-2 bit sequences, too. As expected, a larger free distance of rate 1/3 codewords brings a better decoding performance. For example, the BER is about 10^{-5} at 1.5dB, while

in the Berrou's example the BER is about 10^{-5} at 2dB. Similarly, by computer search we calculate the free distance asymptotic performance as

$$P_b \approx \frac{16272}{4096} Q \left(\sqrt{24 \frac{2E_b}{3 \times N_o}} \right). \quad (3.2)$$

It can be concluded from the figure that the decoding performance curve is quite close to the asymptotic performance after its waterfall region. We can use this free distance asymptotic performance to estimate the decoding performance for this CCSDS encoder.

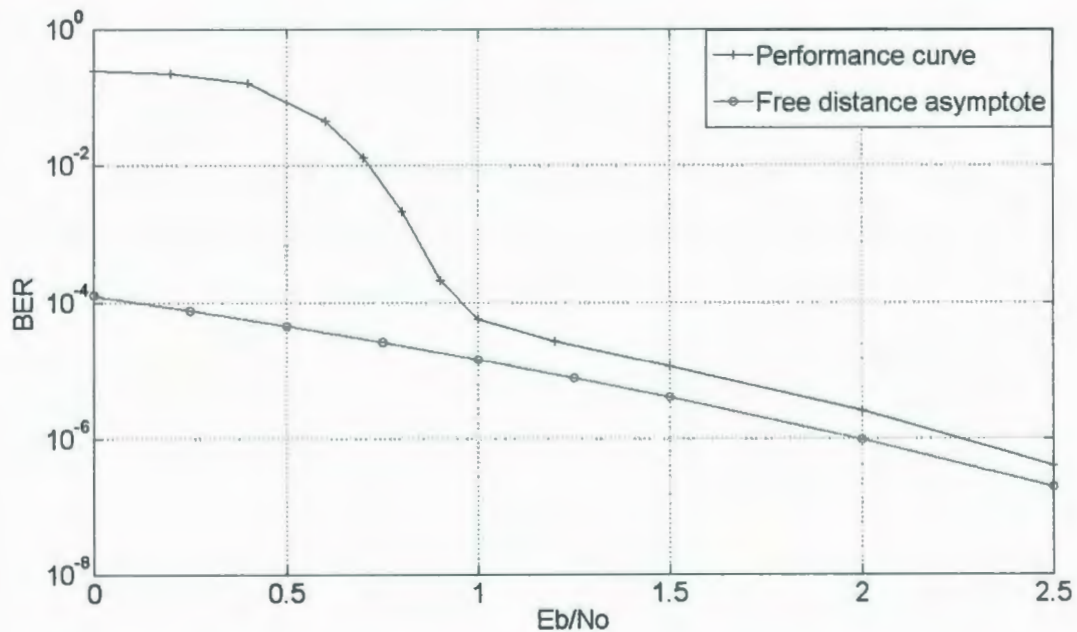


Figure 3.4: Decoding performance and free distance asymptotic performance of the CCSDS encoder

3.3 Limited Bits Representing a Number

So far, we have discussed the architectures of both turbo encoders and turbo decoders, explored the properties that contribute to the decoding performance at a given BER, and investigated two turbo codes with their free distance asymptotic performance. Some key components, for example, the MAX-LOG-MAP algorithm are explained in detail. We

have also investigated the decoding performance of other turbo codes with various E_b/N_o . Those investigations are carried out by software simulations in Java.

However, these discussions and considerations are limited because they are not based on the hardware implementation, except the sliding window technique. Our goal is to implement a turbo decoder in hardware devices, for example, on FPGA boards. We need some extra considerations to implement a software model in hardware. In this section, we discuss a basic issue in the hardware implementation: the data format used in implementing a turbo decoder.

In software simulation, we employ a data type to store a number for operations. For example, if "1" and "2" are store in "double", then the sum "3" is generated in "double". Those number can be easily converted to "int" without extra considering any change in hardware, even if a "double" is stored in 64 bits and an "int" is stored in 32 bits in machines. But in hardware implementation, the adders and memory storages need to be updated if the data format is changed. For example, a 32-bit register cannot save a value of "double" without any loss, while a 64-bit register saves a value of "int" with redundancy. Therefore, before we implement the turbo decoder efficiently, we need to consider how many bits we should use to represent a number.

There are two requirements in determining the limited bits to represent a number. First, any computation loss due to the limited bits should be controlled in a reasonable level. We have to confirm the decoding correctness. Second, the redundancy should be as little as possible. The redundancy often means a waste of hardware resources. Then, we can establish software model to simulate the hardware implementation. Note that all number are represented in 2's complement in many machines.

Given a limited number of bits representing the numbers in a computation, there are computation errors if an overflow happens. To avoid the computation errors, the bits are supposed to be enough to accommodate the maximum and the minimum value in decoding. In other words, the maximum and the minimum values occurring in the decoding process determine necessary bits for numbers. Note that the forward metrics become larger and

larger when the number of decoding stages increases because of the $\max()$ function in each stage. Therefore, metrics quantization is applied to keep the metrics small in each window, if the sliding window is applied. That means, the forward metrics are reduced in each window by subtracting the minimum values among them, before they are delivered to the next window.

In [19] and [20], theoretical upper bound of those internal metrics dynamic range was derived for MAX-LOG-MAP algorithm. It was shown that this range depends on the number of states of the constituent encoders, SNR, and the decoding iterations. In [19], quantization is executed by subtracting the maximum values among the metrics. In [21], it was proposed that at least 9 bits for the integer part if the encoder has 16-states and at an SNR of 5dB.

We employ Monte Carlo methods to investigate the limited bits to represent a number in turbo decoding. The limited bits are supposed to be two parts, the integer and the fraction part, with a decimal point between them. In this section, the notation (x,y) is used to represent x integer bits and y fraction bits. In general, x gives the range of the numbers and y represents the computing precision.

3.3.1 Limited Bits Representing a Number without Sliding Window

In our design, the constituent RSC encoders are derived from the rate 1/2 convolutional encoder $(37, 21)_{(8)}$. The notation $(37, 21)_{(8)}$ represents the generator polynomials in octal form. The interleaver is a block interleaver, with a block size of 4096. We take a fixed iteration number of 10 as the stopping criterion. After 10 iterations, the likelihoods are generated, and then we make a hard estimate to determine whether the transmitted bit is '0' or '1'.

We try to find the minimum number of bits for numbers without significant performance degradation when the sliding window is not applied. As explained before, the minimum number of bits is determined by the maximum value in decoding.

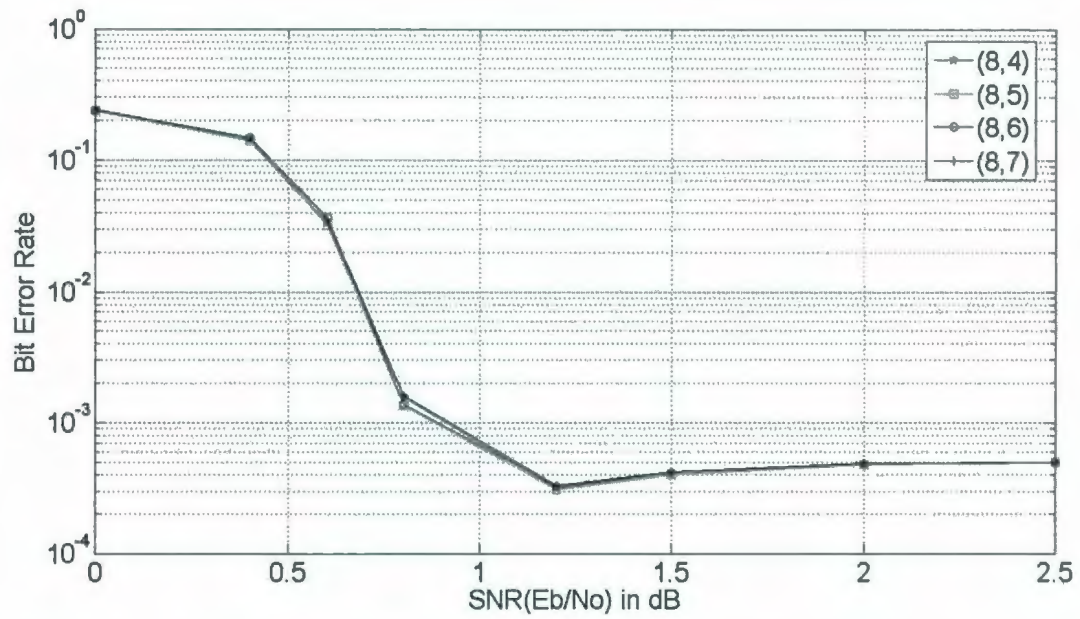


Figure 3.5: Decoding Performance with 8 integer bits, without the sliding window

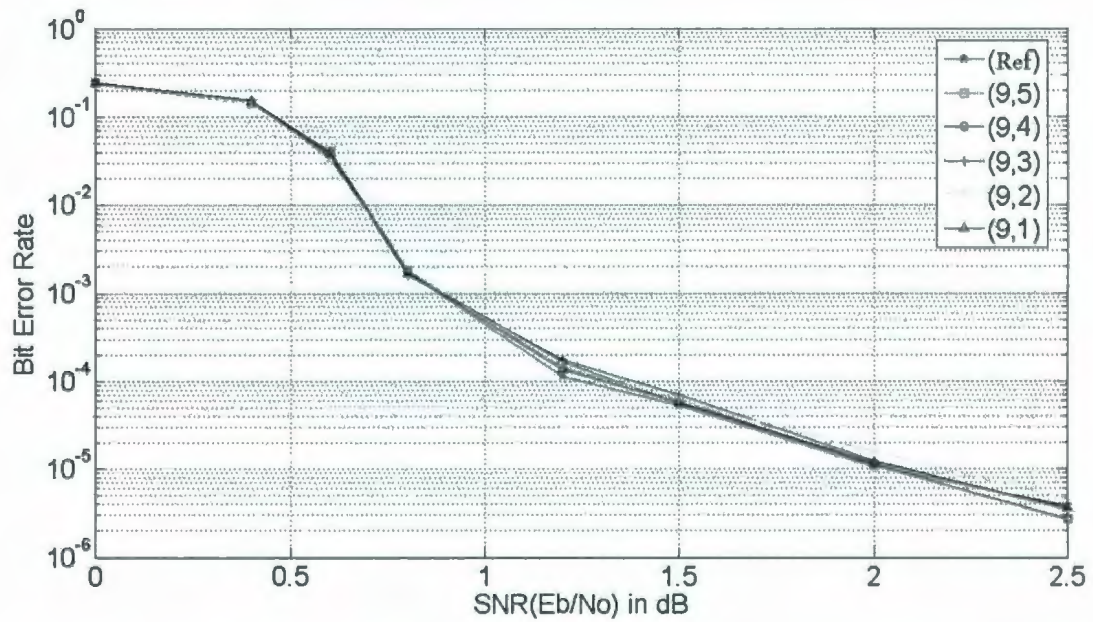


Figure 3.6: Decoding Performance with 9 integer bits, without the sliding window

Figure 3.5 shows the decoding performance when the number of integer bits is 8. At high SNRs, the BER increases slowly. It is explained that the metrics become large when the SNR increases. These extra bit errors are regarded as significant degradation in performance. So it is not accepted and more integer bits are needed. Figure 3.6 shows the decoding performance when the number of integer bits is 9. This figure shows that the decoding performance does not degrade compared with reference curve. When the number of bits for the fraction part decreases to 1, the BER rate is still low enough. So we accept that the minimum bits for integer and fraction part are 9 and 1, respectively.

3.3.2 Limited Bits Representing a Number with Sliding Window

In this section we employ a sliding window technique. Compared with the scheme when it is not employed, we found that the branch metrics and forward metrics are not changed, because we can load the forward metrics of the first stage from the previous window. But the backward metrics are changed, because the computation of backward metrics starts from the last stage of each window, and without the sliding window the computation of backward metrics starts from the last stage of the block. The backward metrics are not accumulated. Since the size of a window is far less than the size of a block, obviously it becomes smaller, so we do not need to assign extra bits to them. In Equations 2.9 and 2.11, it can be seen that the backward metrics are dominated by the increasing forward metrics with the increasing index of decoding stage. So it is not necessary to store the APP L-values or extrinsic probabilities with more bits. Finally, we draw a conclusion that the minimum number of bits, which we got without the sliding window, does not cause significant computation errors when the sliding window is employed. It is (9,1).

After the number of bits is determined, we investigate the decoding performance with different sizes of window, with various number of bits released. In Figures 3.7, 3.8, 3.9, 3.10 and 3.11, the number of released bits ranges about from 4 to more than a half of the window size. Compared those figures with Figures 3.5 and 3.6, it can be seen that the

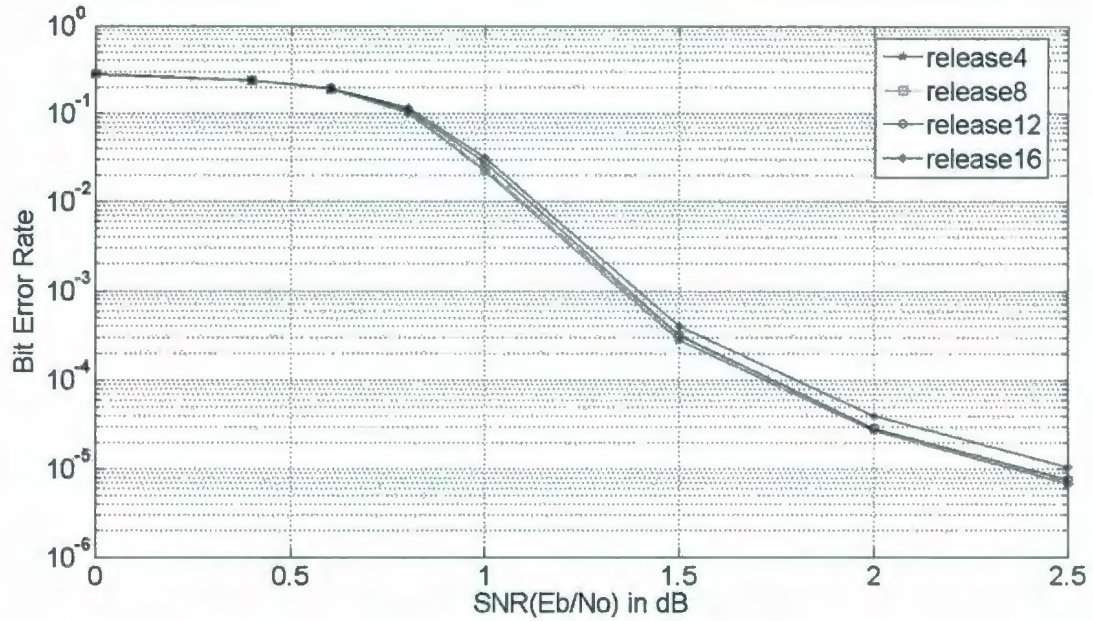


Figure 3.7: Decoding performance with a 32-bit sliding window, 9 integer bits and 1 fraction bit

decoding performance when sliding window is not employed is about 0.5 dB better than the cases when it is employed, because of the inaccuracy of the backward metrics.

From these figures, it can be seen that the decoding performance is acceptable, not significantly degraded, for different window sizes. In Figure 3.7, the BER is a little bit higher if 16 bits are released. In the Figure 3.8 and 3.9, we also find that the BERs become a little bit higher when the number of released bits is larger than the half size of the window. In Figure 3.10 or 3.11, the performance curves are almost same with various number of bits released. That is to say, when the window size is not large, it suggests that the maximum number of bits released should be less than or equal to the half size of the window. When the window size increases, for example, to 80 or 128, the safe number of bits released in each window can be larger than a half of the window size.

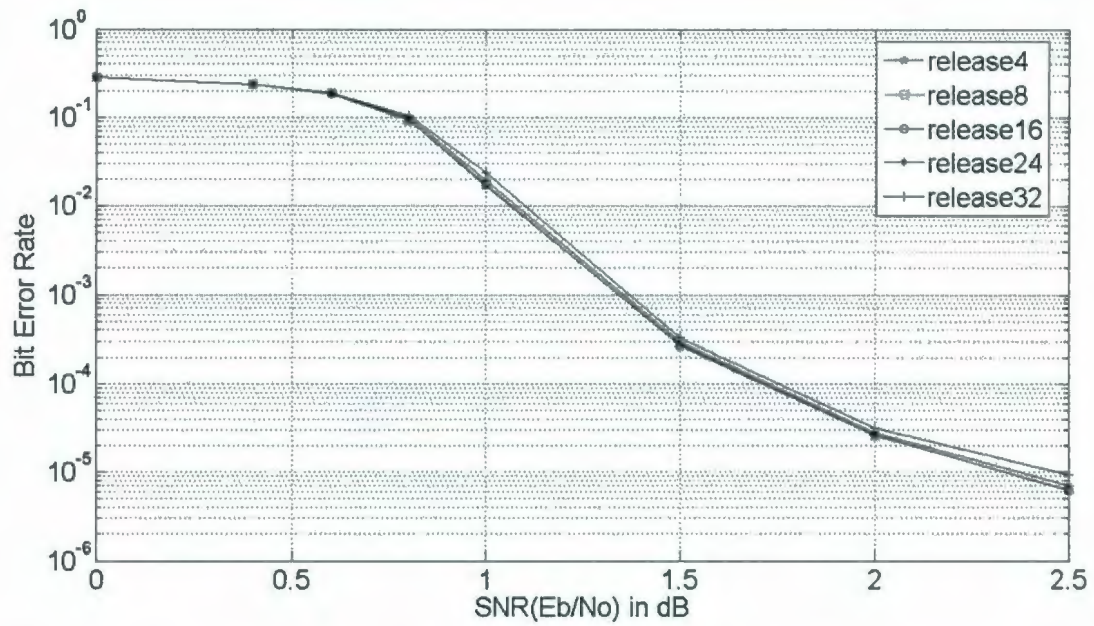


Figure 3.8: Decoding performance with a 48-bit sliding window, 9 integer bits and 1 fraction bit

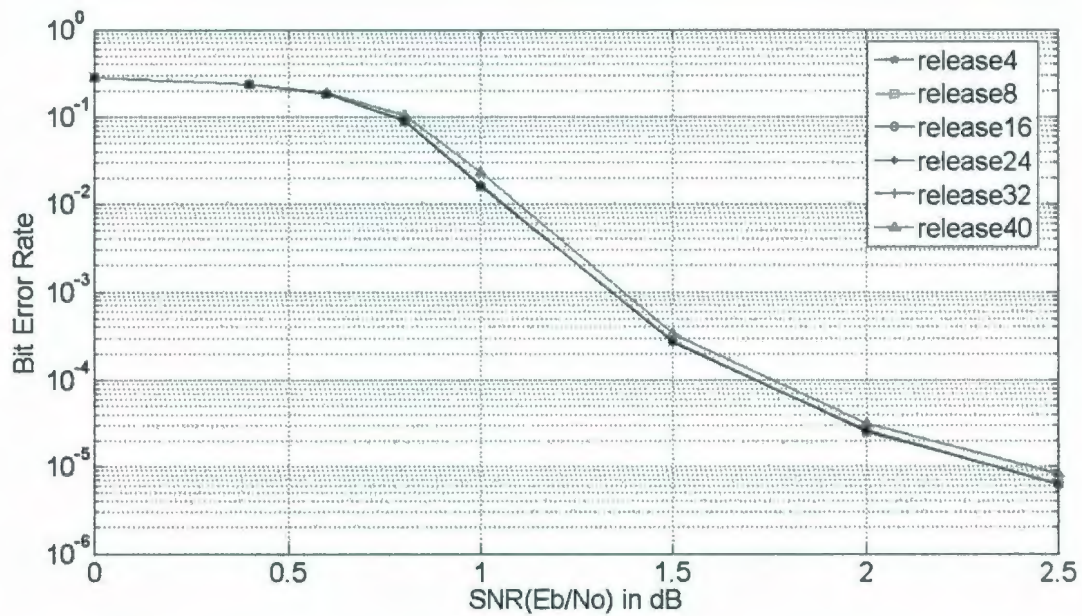


Figure 3.9: Decoding performance with a 64-bit sliding window, 9 integer bits and 1 fraction bit

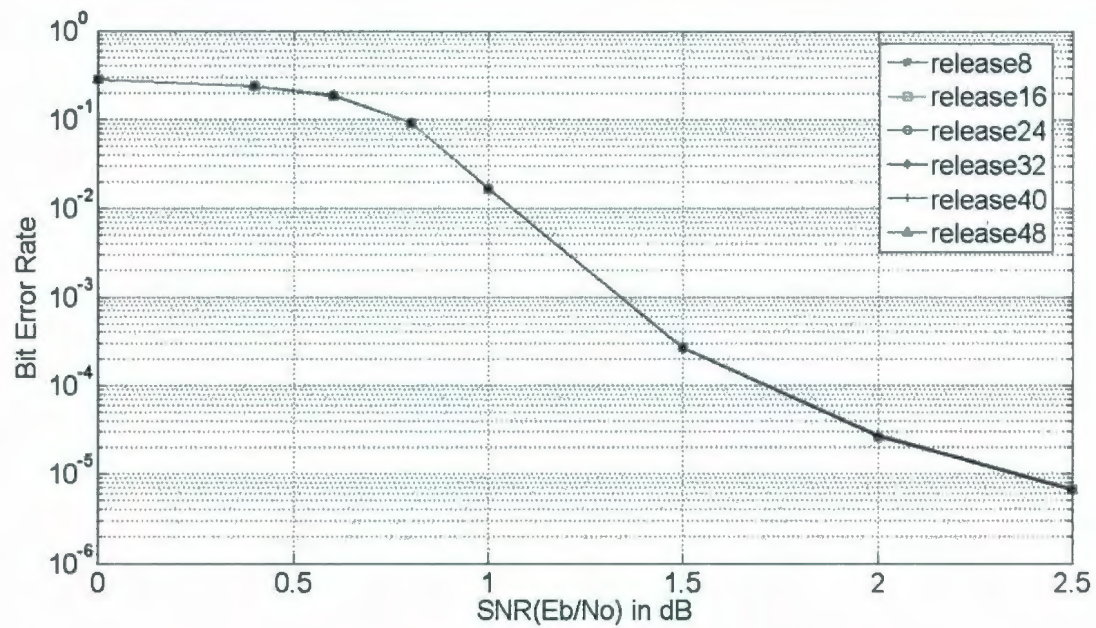


Figure 3.10: Decoding performance with a 80-bit sliding window, 9 integer bits and 1 fraction bit

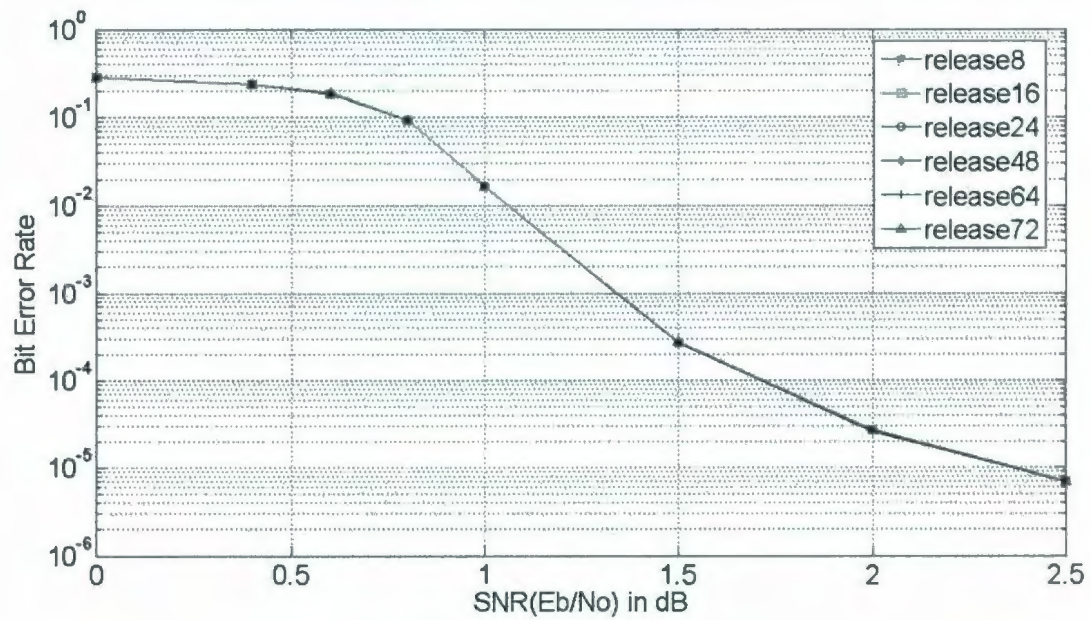


Figure 3.11: Decoding performance with a 128-bit sliding window, 9 integer bits and 1 fraction bit

3.4 Summary

In this chapter, the performance of turbo codes is investigated by using Java simulations. First a general performance of a 16-state turbo code is provided. Then, the free distance asymptote of a turbo code is compared with its decoding performance. It helps us to deeply understand error floor phenomenon in a turbo code. A hardware consideration, the limited bits representing a number, is proposed as a fundamental issue in hardware implementation. This consideration is further investigated by using Java simulations. It is shown that a decoder using a 32-bit sliding window, 9 integer bits, and 1 fraction bit produces acceptable performance the 16-state turbo code.

Chapter 4

Hardware Implementation of a Pipelined Turbo Decoder

In the previous chapter, we investigated the decoding performance of turbo codes in software simulations. Although the Monte Carlo method employed in software simulations provide us decoding performance of turbo codes, the design in software can be only viewed as a model because physical systems are not considered in detail. Therefore, hardware implementation on FPGA board is necessary to verify the design. In this section, we will discuss the hardware implementation of a 16-state turbo decoder on an FPGA board, because it generally outperforms a 4 or 8 states turbo decoder in terms of BER. This FPGA implementation is characterized by the BER, the number of gates, and the decoding delay.

We verify the BER by comparing waveforms with the data from simulations. The BER is expected to be equal to that in the software model, when the output signal waveforms from the FPGA board are exactly the same as what we got from the software simulations. Consequently, we can say the decoding correctness in the FPGA implementation is confirmed. The number of gates and the decoding delay for a certain algorithm has no relationship to software model. They depend on the hardware design. The gates in circuit are the cost, while the decoding delay represents the efficiency. In general, the decoding delay

becomes smaller if more gates are employed in the circuit and more processes are computed in parallel. Therefore, there is a trade-off between the hardware size and decoding latency, the cost and efficiency.

The hardware implementation is completed step by step. First, the MAX-LOG-MAP algorithm is implemented in a Virtex 5 board. The Virtex 5 board provides abundant resources, for example, the *random access memory* RAM blocks. At this stage, hardware optimization is not considered yet. The decoding correcting correctness is the most important issue. The algorithm is not modified in the computational order and the memory blocks for metrics are not saved. Second, the MAX-LOG-MAP algorithm implementation, which is viewed as an SISO decoder, is developed to a turbo decoder. According to Figure 2.3, it can be seen that in hardware implementation only one SISO decoder is required due to the identical architecture in both SISO decoders. A block of memory is added as an interleaver/deinterleaver. A more complex controller is employed to realize the sliding window technique and the iterative decoding process. Third, this hardware implementation is improved. The number of memory blocks are reduced by altering the computational order. In the MAX-LOG-MAP algorithm, the various metrics are calculated in a pipelined way so that the decoding latency is significantly reduced. Finally, the resources utilization is considered. The pipelined turbo decoder is implemented in other cheaper FPGA boards. Although the implementation in those boards employs different components, the main architectures of the MAX-LOG-MAP algorithm and the turbo decoder are not changed. However, the cheaper boards have fewer resources and thus the utilizations of those FPGA devices become higher. The FPGA implementation is supposed to be optimized when the utilization of on-board components reaches 100%. In this thesis, only the output signals and the waveforms from the Virtex 5 board are used in further discussion and analysis. The signals and the waveforms from other boards are mostly similar, but not identical.

4.1 Xilinx XUPV5-LX110T Evaluation Platform

The XUPV5-LX110T Evaluation Platform, which includes a Virtex-5 FPGA device, is available to educational institutions through the Xilinx University program. It is a unified platform for teaching and research in disciplines such as digital design, embedded systems, digital signal processing and communications, networking, video and image processing, and so on. It features two Xilinx XCF32P platform flash PROMs, Xilinx SystemACE compact flash configuration controller, 10/100/1000 tri-speed Ethernet interfaces, USB host and peripheral controllers, stereo AC97 codec, RS-232 port, 16x2 character LCD, and many other I/O devices and ports [22]. The appearance of the XUPV5-LX110T board is shown in Figure 4.1. We develop our design on this board.

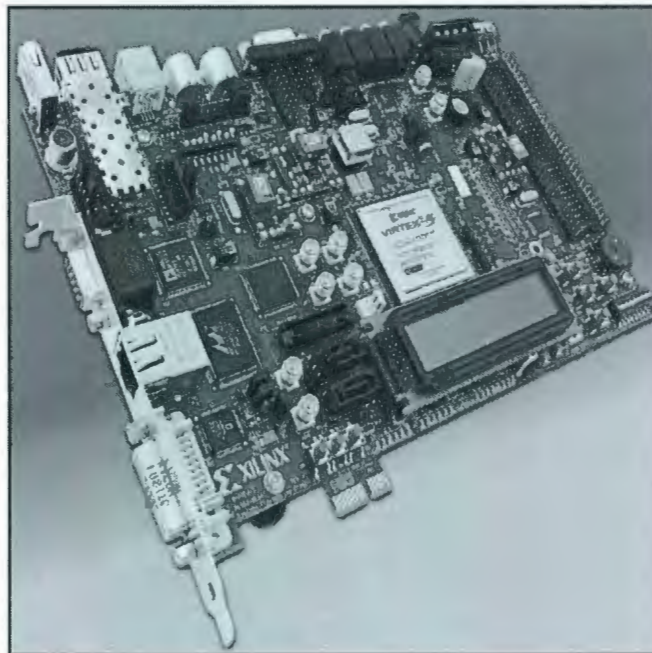


Figure 4.1: XUPV5-LX110T FPGA board

The software to implement the design and program the board is Xilinx Modelsim and ISE, respectively. First, the design is coded in VHDL and simulated in Modelsim. Second, the .bit file is generated and programmed in the FPGA board by Xilinx ISE. The intermediate signals are analyzed by using ChipScope and the output signals are displayed

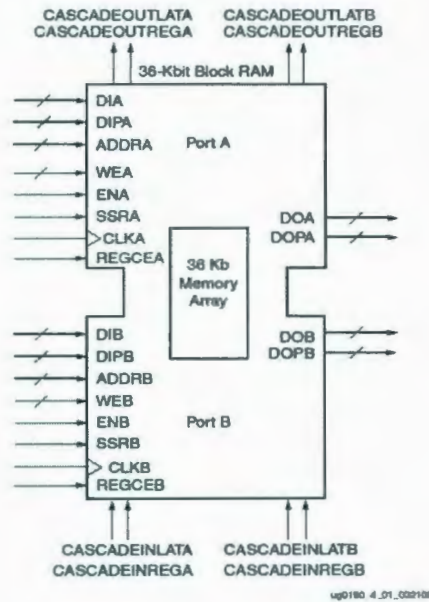


Figure 4.2: A true dual-port block RAM

in a hyper-terminal window. Finally, we verify the decoding correctness by comparing the waveforms with data in the simulation.

There are two main on-board primitives we employ in our design. The first is the block RAM that is used as a memory for decoding metrics. Figure 4.2 [23] is the data flow of a synchronous 36Kb block memory with a true dual-port A and B. Data can be written to or read from either or both ports. Each write operation is synchronous, and each port has its own address, data in, data out, clock, clock enable, and write enable. The read and write operations are synchronous and require a clock edge [23].

The second is the DSP48E slices, which are designed as addition, multiplication, and bit logic operations units. These slices are specially designed for operations in DSP when they are cascaded. Figure 4.3 [24] shows the architecture of a DSP48E slice. These slices can be configured as 25 x 18 multipliers and an add/subtract function that has been extended to function as a logic unit. This logic unit can perform a host of bitwise logical operations when the multiplier is not used. The DSP48E slice includes a pattern detector and a pattern bar detector that can be used for convergent rounding, overflow/underflow detection

for saturation arithmetic, and auto-resetting counters/accumulators. The document [24] provides the specifications of these slices.

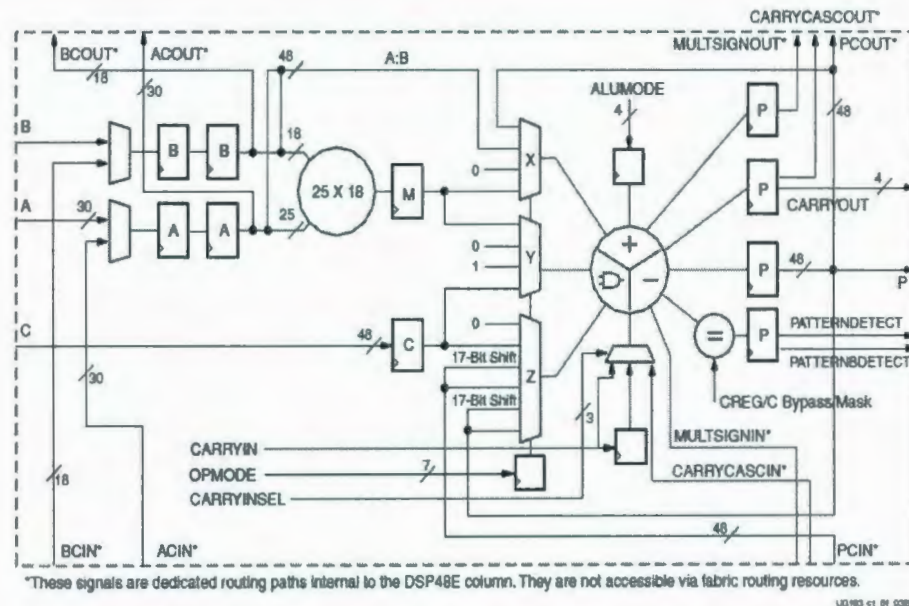


Figure 4.3: Virtex-5 DSP48E slice

Xilinx XUPV5-LX110T board provides various interfaces, for example, a RS232 port. The RS232 serial protocol is simple and widely used, especially in the applications that do not demand very high speed communication. With a hyper-terminal window the communication through RS232 port can be monitored. We connect the on-board RS232 port to a desktop through a cable. The baud rate of the communication is set to 25, a speed slow enough to read. Finally, the output can be displayed on the hyper-terminal.

4.2 Hardware Considerations of Turbo Codes

We focus only on the design of turbo decoders in our research, not the encoders, because the encoders are comparatively simple. The encoder is composed of interleaver/deinterleaver and RSC encoders. Both the encoder and the corresponding decoder employ the identical interleaver and deinterleaver, which is implemented by a block of memory. In general,

the RSC encoders are constructed from a few flip-flops or registers. But the decoders are further characterized by the complex decoding algorithm.

In software simulations, metrics in the MAP algorithm are stored in “double”, which accommodates for numbers in a very large range. The data “double” is of high accuracy. But this representation of “double” is regarded in hardware implementation as an unlimited number of bits. It is a redundancy for our design in hardware. Therefore, first of all, we need to determine the data type applied in memories and arithmetic units. As explained before, a limited number of bits are used to represent a number. All data are represented in 2’s complement. A number is set to the maximum or the minimum. That is to say, it is saturated when an overflow or underflow happens.

The second consideration is to implement the design in a combinational or a sequential logic circuit. More combinational logic makes the implementation simpler, at cost of more gates and a longer latency for the longest path in circuit. To sketch the circuit, we need to estimate the numbers of additions and multiplications. From the Equation 2.4, it can be seen that there are 32 branches in a stage and two multiplications and two additions for each branch. It would be at least 32 sets of branch metrics calculations in each stage, with each set containing two multipliers and two adders, if all additions and multiplications work in parallel for a 16-state decoder. In this way, the hardware implementation consists of many duplicated calculations and the hardware size is high. On the other hand, if the circuit is implemented fully in sequential logic, the decoding latency may be prohibitive. Assuming a Booth encoder completes one 16X16 multiplication in 32 clock cycles (at least 1 clock cycle for loading and 1 clock cycle for shifting), it is necessary to compute one branch metric in 64 clock cycles and compute all branch metrics in one stage at least $32 \times 64 = 2048$ clock cycles, even if we do not consider forward and backward metrics. Our pilot design shows that the decoding latency is very long if the Booth encoder is used. Fortunately, the Xilinx XUPV5-LX110T Evaluation Platform provides us the DSP48E slices, which can be configured as embedded multipliers. With the help of 2 embedded multipliers it takes 1 clock cycle for one branch metric and 17 clock cycles for all branch metrics in a stage

(including 1 clock cycle for the initialization). In this scheme, the latency is significantly reduced from 2048 clock cycles to 17 clock cycles. Therefore, the metrics calculators are implemented in sequential logic, while 2 DSP48E slices are employed.

Finally, we set some parameters to a small number to simplify the design. The number of iterations is assumed to be 2. The interleaver size is 64 and window size is 16, so that the number of windows in a block is 4. Here we employ 16 bits, 13 bits for the integer part, 2 bits for fraction bits, and 1 bit for the sign. When we use a small number of iterations and small blocks, we can save a lot of time in verifying the data and waveforms. The decoder can be configured to a 16-iteration decoder by replacing a 2-bit counter with a 4-bit counter without other modifications. The essential issue in hardware design is to confirm the decoding correctness. That is to say, the software simulation results should be completely verified by hardware implementation. The verification strategy is illustrated in Figure 4.4.

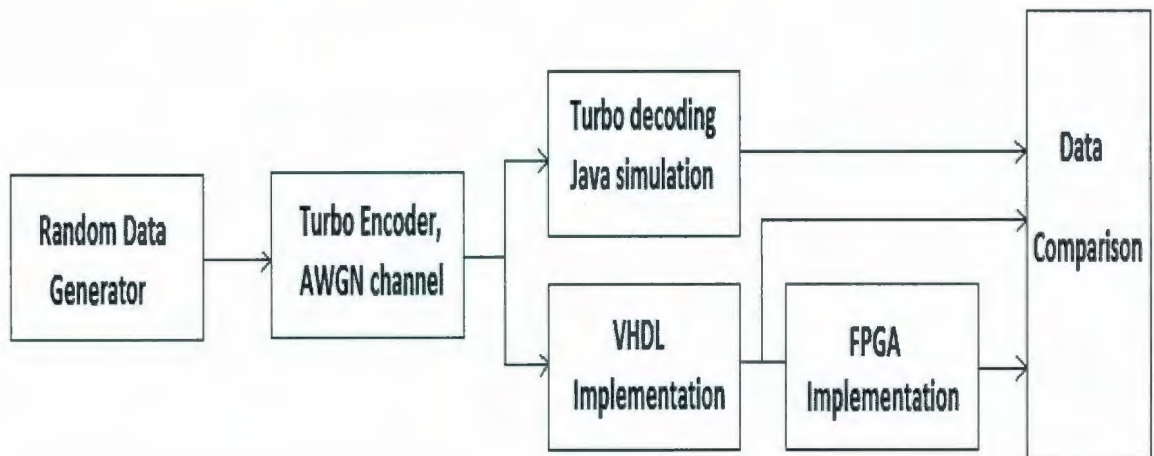


Figure 4.4: Verification strategy

4.3 FPGA Implementation of a Pipelined Turbo Decoder

4.3.1 Implementation of the MAX-LOG MAP algorithm

First, we consider the MAP algorithm employed in the SISO unit. The algorithm can be modified to *log* domain as the MAX-LOG-MAP or the LOG-MAP, because the exponential operations are introduced in the MAP algorithm. The difference between the MAX-LOG-MAP and the LOG-MAP is illustrated by Equation 2.7 and 2.8. The second item on the right hand can be approximately estimated in segments and implemented by a look up table in the hardware implementation. In our design, the MAX-LOG MAP is employed.

The MAX-LOG-MAP algorithm is still a complex algorithm with considerable intermediate metrics, such as forward metrics, backward metrics, and branch metrics. The total number of those metrics is $(2 + 1 + 1) \times 2^M \times N$, with M as the memory length of the turbo encoder, and with N as the length of a block or a window. It is in proportion to the block length N and has exponential relation to the encoder memory length M . Obviously, more operations than this number are necessary to obtain those metrics. For a 16 state decoder, there are at least 2×32 multiplications and 2×32 additions for branch metrics, 2×16 additions for forward metrics, 2×16 additions for backward metrics, and 2×32 additions to obtain one likelihood value. The numbers of multiplications and additions are $2^{(M+2)} \times N$ and $3 \times 2^{(M+2)} \times N$, respectively. Here we do not include the comparisons between two numbers. In [21], the authors estimated the computational complexity of different decoding algorithms, as shown in Figure 4.5. The MAP algorithm is not included due to its inherent complexity for practical turbo decoder implementation. All operations include addition, max, table lookup and inversion. The LOG-MAP has table lookup operations, and about 1.33 times of additions as the MAX-LOG-MAP. The MAX-LOG-MAP has approximately 3 times of additions and 2 times of max operations in the SOVA algorithm when M is large enough. In [25], the author improved the MAP algorithm by eliminating the intermediate

Operations	Add	Max	Table Lookup	Inversion
Log-MAP	$16 \cdot 2^M - 1$	$4 \cdot 2^M - 2$	$4 \cdot 2^M - 2$	3
Max-Log-MAP	$12 \cdot 2^M + 1$	$4 \cdot 2^M - 2$		3
SOVA	$4 \cdot 2^M + 6 \cdot M + 14$	$2 \cdot 2^M + 6 \cdot M + 6$		2

Figure 4.5: Computation complexities of different decoding algorithms

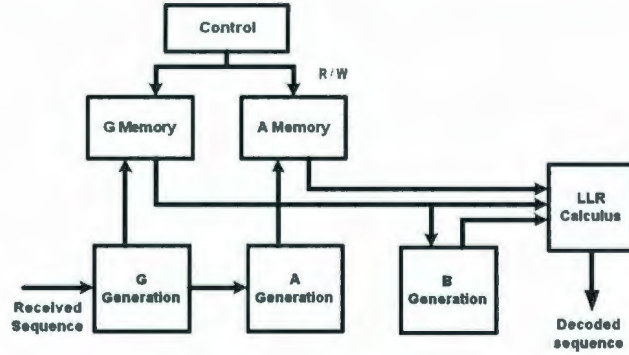


Figure 4.6: Architecture of the MAP/LOG-MAP/MAX-LOG-MAP algorithm

metrics. The author estimated the numbers of multiplications and additions as $2^{M+N} + N$ and $(2^M + N - 1)2^N - 2N$. It can be inferred that in the MAP algorithm the additions and multiplications numbers have exponential relation to N .

The block diagram of a typical implementation of MAP/MAX-LOG-MAP algorithm is shown in Figure 4.6 [26]. In this figure, “G” represents the branch metrics, “A” represents the forward metrics and “B” represents the backward metrics. In this design, the memories for branch and forward metrics are dispensable. The blocks of “G Generation”, “A Generation”, “B Generation”, and “LLR Calculus” can be implemented by either combinational or sequential logic. The controller is responsible for loading and storing the metrics. The order of the metrics computation is G, A, B, and LLR. First, the branch metrics are calculated. Second, starting from the beginning of each block, the SISO decoder calculates the forward metrics and stores them in the memory. At the end of the forward metrics calculation, the decoder starts backward metrics calculation in the backward direction, as well as LLR calculation. This computation order conforms to the principles of MAP algorithm discussed in Equations 2.4, 2.5, 2.6, and 2.9.

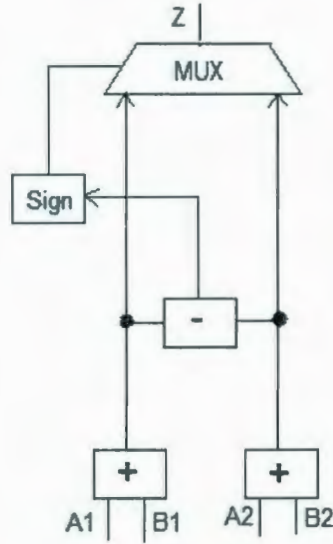


Figure 4.7: Architecture of ACSU in the MAX-LOG-MAP algorithm

When forward and backward metrics are calculated, *addition-comparison-selection units* (ACSU) are employed. They are employed in the blocks of “G Generation”, “A Generation”, “B Generation”, and “LLR Calculus”. Figure 4.7 and 4.8 illustrate the ACSUs in the LOG-MAP and MAX-LOG-MAP algorithm, where “LUT” denotes the look up table, and the output z is equal to $\max(A1 + B1, A2 + B2)$ and $\max(A1 + B1, A2 + B2) + \ln(1 + e^{-|A1+B1-(A2+B2)|})$, respectively. Similar architectures of ACSU can be found in [27].

The backward metrics is always calculated recursively from the last stage of either a window or a block. When the sliding window technique is used, we release the first several bits, not the last bits. According to Equation 2.9, we have to calculate all backward metrics in order to release the first bit. For the last bits unreleased, it is not necessary to calculate those intermediate forward metrics. Therefore, we change the computation order by calculating backward metrics first. Figure 4.9 shows the dataflow of metrics in our design. A *finite state machine* (FSM) is employed as a controller to coordinate the blocks in the block diagram. In this figure, the memories for branch and backward metrics are constructed by RAMB18 that is introduced before. Since every metric is a 16-bit number, the same width as the input of a RAMB18 block, to load and store 16

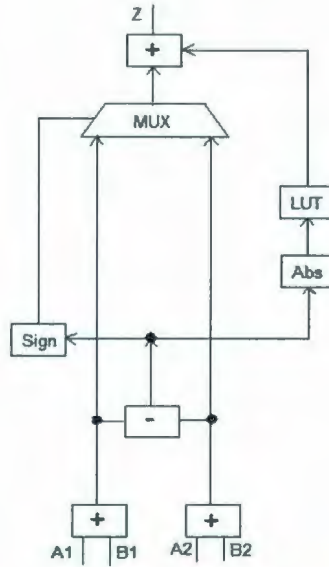


Figure 4.8: Architecture of ACSU in the LOG-MAP algorithm

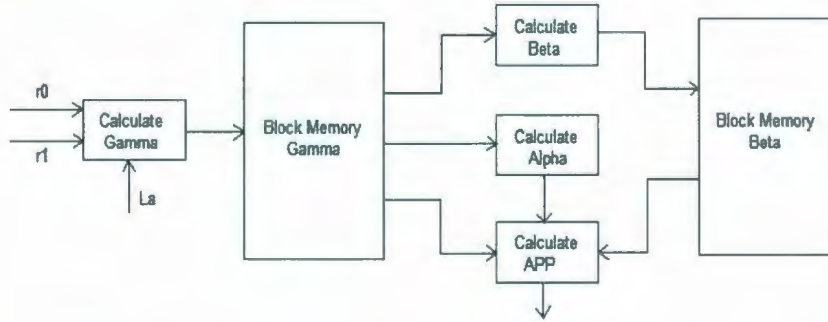


Figure 4.9: The dataflow of the MAX-LOG-MAP algorithm

beta metrics simultaneously need 16 RAMB18s to work as a group, as shown in Figure 4.10. The advantage of this scheme is that 16 numbers are read or written in one clock cycle. But the disadvantage is that the utilization of RAMB18s is low. For example, if a received block of data is a sequence of 32 signals, the utilization of each RAMB18s is $32/1024 = 0.03125$ for branch metrics and backward metrics. That's the trade-off. Fortunately, the XUPV5-LX110T board supports loading and storing metrics in parallel by providing 296 RAMB18s [28]. We employ 32 RAMB18s for branch metrics and 16 RAMB18s for beta metrics, considering that the loading and storing metrics are very frequent operations in the MAX-LOG-MAP algorithm.

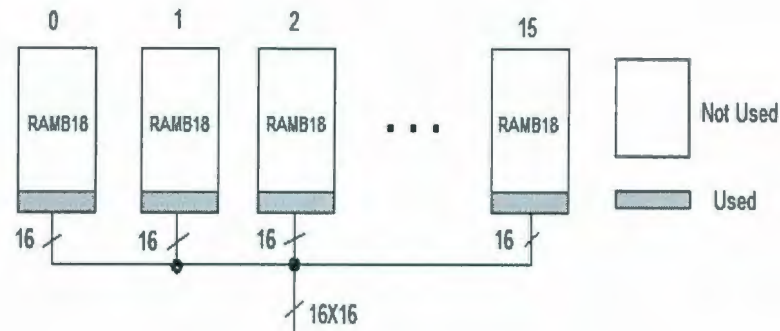
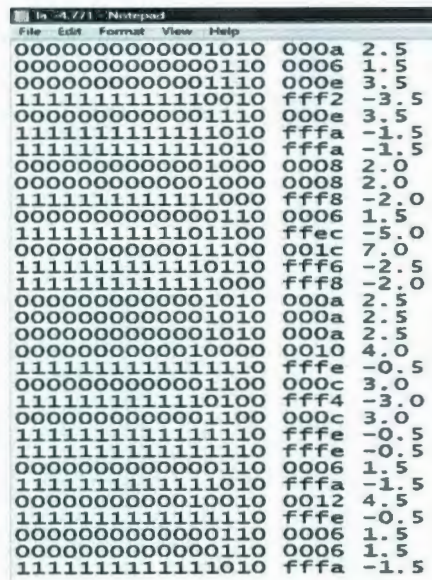


Figure 4.10: the RAMB18 group

In Java simulations, the additions are saturated as if limited bits are used to represent a number. When a number needs more than 16 bits, an overflow or underflow happens. This number will be replaced by the maximum value $7FFF_{(16)}$ or the minimum value $8000_{(16)}$, with 16 in the subscription as a hexadecimal number. In a $16X16$ multiplication, the result is a 32-bit number with the decimal point locating in the fourth bit from the LSB. This number is converted to a 16-bit number with the decimal point locating in the second bit from the LSB. This conversion causes some computation errors because some tail bits are discarded. The computation errors are accumulated during the decoding. It is also found that systematic computation errors cannot be completely eliminated when a right shifting operation happens to a limited bits. If a number is divided by 2, the bits are shifted to right and LSB is discarded, whether the LSB is "0" or "1". However, the waveforms of VHDL implementation coincide with the decoded sequence in Java simulations, when both the VHDL implementation and the Java simulation follow the same operations of additions and multiplications. Figure 4.11 provides a block of 32 decoded signals by the MAX-LOG-MAP algorithm in Java simulations. The data in the second column are represented in the form of hexadecimal, which are conveniently used to compare with the waveforms. Figure 4.12 and Figure 4.13 provide the waveforms for the first 8 decoded values. Note that in the small circles there are three variables. "wr_la" is the enable signal to write, "addr_la" is the address of the memory when the data is written, and "app" is the decoded. The first 8 decoded words can be read from the waveforms. They are "000A", "0006", "000E",

“FFF2”, “000E”, “FFFA”, “FFFA”, and “0008”. All 32 decoded words from the waveform are verified. Therefore, the decoding correctness in the MAX-LOG-MAP algorithm is ascertained.



Binary	Hex	Value
0000000000001010	000a	2.5
0000000000000110	0006	1.5
0000000000000110	000e	3.5
1111111111110010	fff2	-3.5
0000000000000110	000e	3.5
1111111111110101	fffa	-1.5
1111111111110101	fffa	-1.5
0000000000001000	0008	2.0
0000000000001000	0008	2.0
1111111111111000	fff8	-2.0
0000000000000110	0006	1.5
1111111111110100	ffec	-5.0
0000000000001100	001c	7.0
1111111111110110	fff6	-2.5
1111111111111000	fff8	-2.0
0000000000000101	000a	2.5
0000000000000101	000a	2.5
0000000000000101	000a	2.5
0000000000001000	0010	4.0
1111111111111110	fffe	-0.5
0000000000000110	000c	3.0
1111111111110100	fff4	-3.0
0000000000000110	000c	3.0
1111111111111110	fffe	-0.5
1111111111111110	fffe	-0.5
0000000000000110	0006	1.5
1111111111110101	fffa	-1.5
0000000000001001	0012	4.5
1111111111111110	fffe	-0.5
0000000000000110	0006	1.5
0000000000000110	0006	1.5
1111111111110101	fffa	-1.5

Figure 4.11: Results of the MAX-LOG-MAP algorithm in Java simulations

The FPGA implementation by the Xilinx ISE optimizes the design automatically, and the architecture of the algorithm is realized by the available primitives on the FPGA board. Besides the VHDL codes, a *user constraint file* (UCF) is necessary in translating, mapping, placing and routing to implement the design. This UCF is to map the input and output signals in the design to specific pins on the FPGA board or use a designated device, for example, the clock. The Xilinx XUPV5-LX110T board provides clocks of various frequencies, 27MHz, 33MHz, and 100MHz. We employ both 33MHz and 100MHz clocks, but it makes no difference in bit errors when implementing this MAX-LOG-MAP algorithm and the turbo decoder.

Figure 4.14 shows a summary of device utilization reported by the Xilinx ISE. Some components, such as registers and flip-flops, are generated automatically. The numbers of them are not controllable unless the VHDL implementation is at the gate level, but they indicate the hardware complexity that can be used to evaluate the implementation

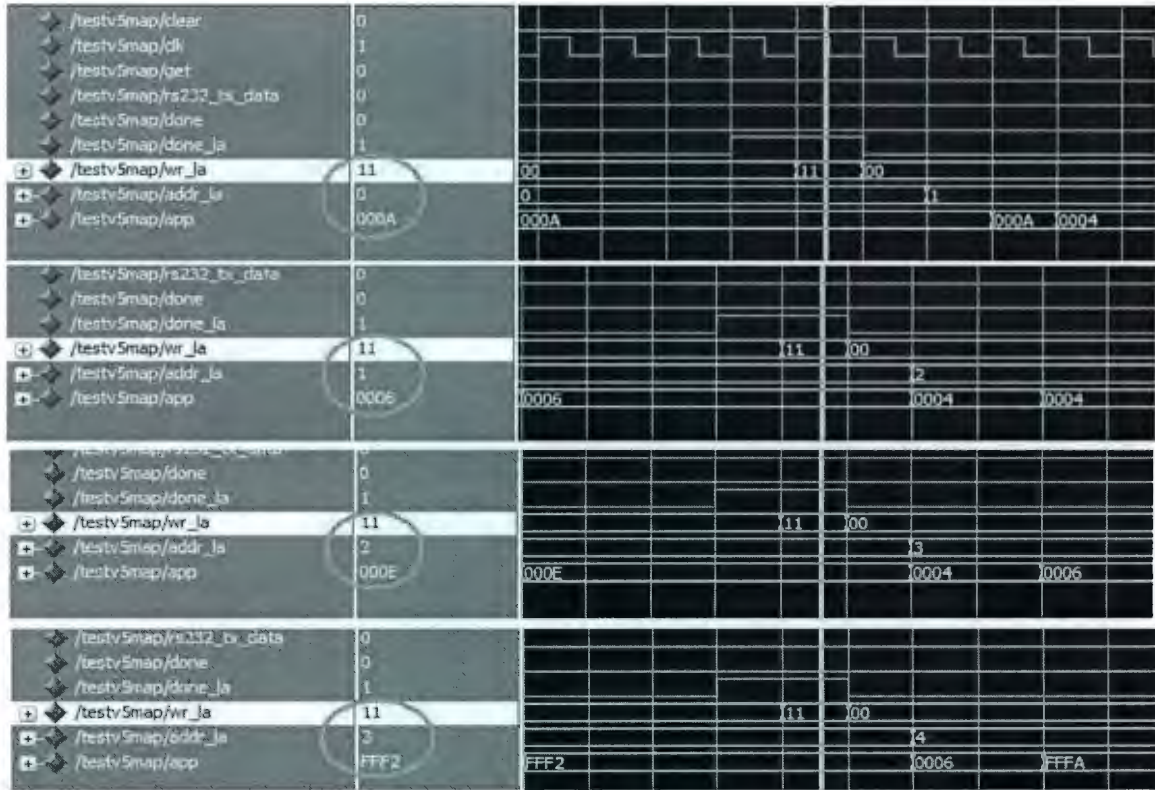


Figure 4.12: Waveforms of the MAX-LOG-MAP algorithm in VHDL implementation (1)

approximately. However, it can be seen that 2 DSP48E slices are used as the embedded multipliers in the implementation. There are a total of 51 RAM18 used as we expected. 32 out of 51 are employed as “Block memory gamma”, 16 out of 51 are employed as “Block memory beta”, and the remaining 3 RAMB18s are used to store the received signals which is not indicated in the block diagram.

4.3.2 Implementation of a Turbo Decoder with the Sliding Window

Figure 2.3 shows the architecture of the iterative decoding in turbo codes. The main components are two SISO units, an interleaver, and a deinterleaver. These two SISO units have duplicated architecture. The interleaver and the deinterleaver have the same structure and inverse functions. Obviously, only one SISO unit and one block memory are necessary in the hardware implementation, as in [21] and [29]. The SISO unit works

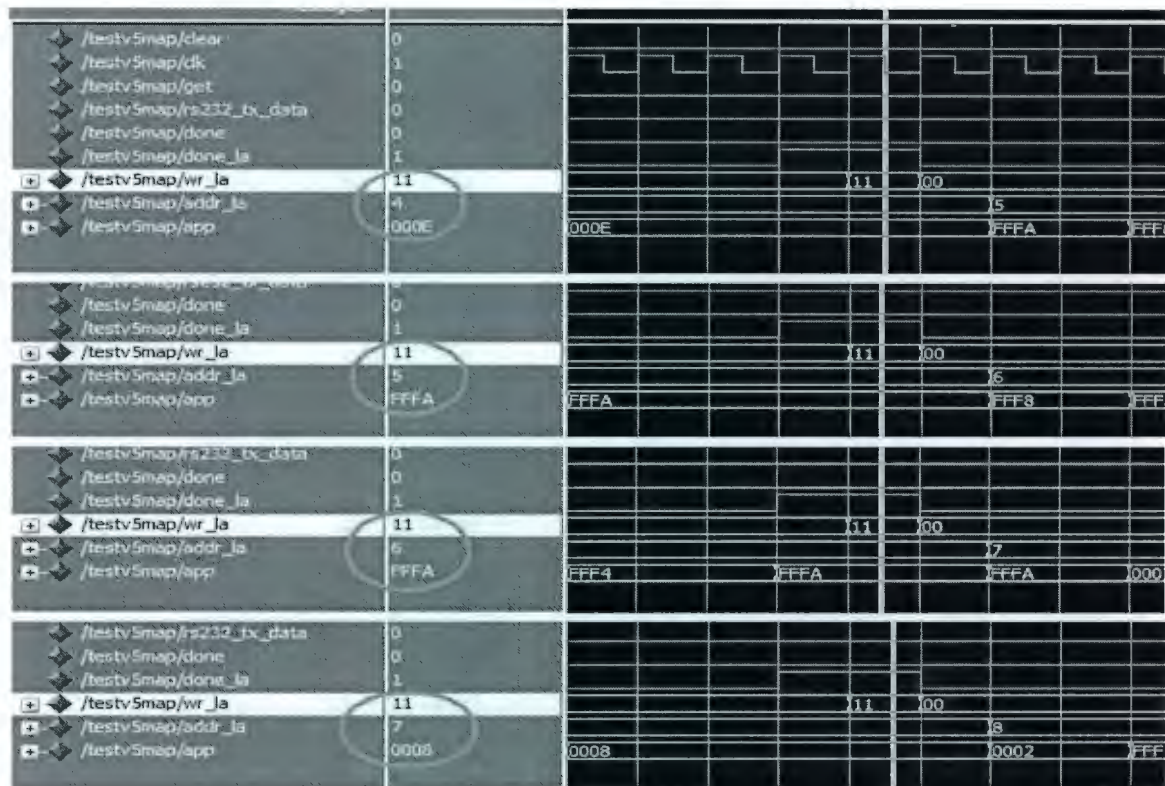


Figure 4.13: Waveforms of the MAX-LOG-MAP algorithm in VHDL implementation (2)

for both upper and lower half iterations. The block memory is designed for both the interleaver and deinterleaver. Figure 4.15 shows the implementation of a turbo decoder based on LOG-MAP in [21]. In this figure, the LLRs are stored in a RAM after they are released. The interleaver and deinterleaver memory stores the mapping addresses, which is used to locate the previous LLR values in the RAM. If the interleaver size is S , then the size of “LLR RAM” is S and the length of decoding in the LOG-MAP decoder is S . In [29], the author proposed a block diagram of turbo codes adapted to standard communications in *wideband code division multiple access* (WCDMA), as shown in Figure 4.16, where “ π/π^{-1} ” denotes the interleaver and deinterleaver functions, “Mem_data” denotes the memory for the received signals, and “Ysl” denotes the memory to store LLR values.

However, these two implementation do not make use of the sliding window technique. The sliding window technique is a good strategy in turbo decoding. It helps to divide a block into many small windows and then decode the entire block. The advantage of this

Device Utilization Summary				
Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	3,014	69,120	4%	
Number used as Flip Flops	3,013			
Number used as Latches	1			
Number of Slice LUTs	4,115	69,120	5%	
Number used as logic	4,048	69,120	5%	
Number using O6 output only	3,876			
Number using O5 output only	51			
Number using O5 and O6	121			
Number used as Memory	59	17,920	1%	
Number used as Shift Register	59			
Number using O6 output only	57			
Number using O5 output only	1			
Number using O5 and O6	1			
Number used as exclusive route-thru	8			
Number of route-thrus	59			
Number using O6 output only	59			
Number of occupied Slices	2,518	17,280	14%	
Number of LUT Flip Flop pairs used	5,850			
Number with an unused Flip Flop	2,836	5,850	48%	
Number with an unused LUT	1,735	5,850	29%	
Number of fully used LUT-FF pairs	1,279	5,850	21%	
Number of unique control sets	887			
Number of slice register sites lost to control set restrictions	2,454	69,120	3%	
Number of bonded IOBs	4	640	1%	
Number of LOCed IOBs	4	4	100%	
Number of BlockRAM/FIFO	38	148	25%	
Number using BlockRAM only	38			
Number of 18k BlockRAM used	51			
Total Memory used (KB)	918	5,328	17%	
Number of BUFG/BUFGCTRLs	2	32	6%	
Number used as BUFGs	2			
Number of BSCANs	1	4	25%	
Number of DSP48Es	2	64	3%	
Number of RPM macros	9			
Average Fanout of Non-Clock Nets	3.22			

Figure 4.14: Device utilization summary of the MAX-LOG-MAP algorithm

technique is that a large SISO unit is not necessary, while the disadvantage is overhead computation because of the overlapping bits in the next window.

To simplify the design and then reduce the workload in data comparison, we set the parameters to small numbers. In the hardware implementation, the iteration number is 2, the block interleaver size is $8 \times 8 = 64$, the window size is 32, the number of windows in a block is 4, and in every window the first 16 bits are released. Figure 4.17 shows the block diagram of the turbo decoder based on the MAX-LOG-MAP with these parameters, where “Memory Data” denotes the memory for the received data from the channel, “Memory Beta” and “Memory Gamma” denote the memory for the backward metrics and the branch metrics

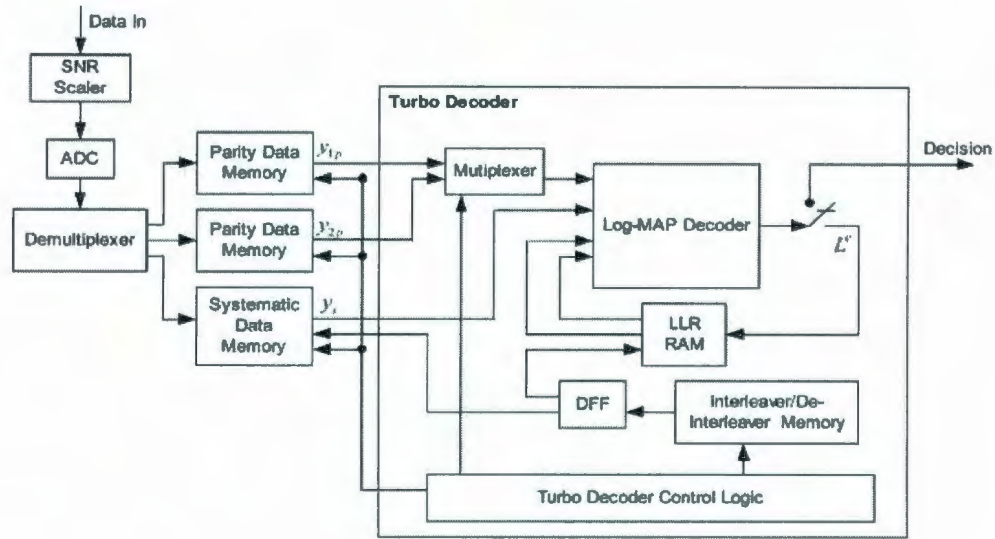


Figure 4.15: Architecture of a turbo decoder (1)

respectively, “Memory Alpha” denotes the memory for the forward metrics, “Iterations C4+” denotes the counter to indicate the half iterations index, “Windows C4+” denotes the counter to indicate the windows index, and “C64+,C64-” and “C32+,C32-” denote the up counters and down counters with the maximum numbers of 64 and 32, respectively. The SISO unit includes the various blocks except the block “Memory Data”, “Memory LLR”, “Memory Alpha”, “Iterations C4+”, “Windows C4+”, “Interleaver/Deinterleaver”, and the controller. Some blocks are introduced by the sliding window technique, such as “Windows C4+” and “Memory Alpha”.

Due to the sliding window technique, we employ the “Memory Alpha” to store the last forward metrics in this window as the initial forward metrics in the next. It has small size, $16 \times 16 = 256$ bits. It may be removed in a future design, if the last forward metrics are retained. We use “Memory Alpha” in order to prevent the data interference between the windows. The block “Interleaver/Deinterleaver” realizes the function and inverse function of interleaver. If the interleaver size is large and the permutation is irregular, then this block can be implemented by a large look up table that stores the mapping addresses, not the data LLR. Small blocks attached to memory blocks, such as “C64+,C64-” and “C32+,C32-

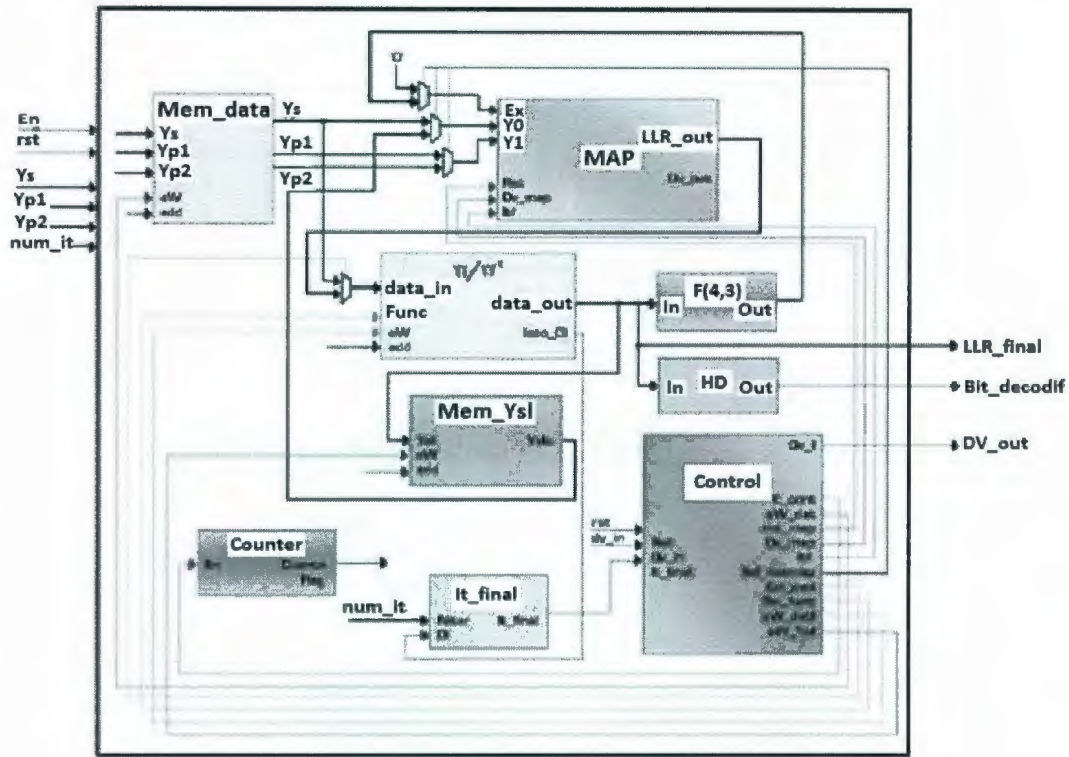


Figure 4.16: Architecture of a turbo decoder (2)

", indicate that these counters are used as address pointers for the corresponding memory blocks. The numbers in the top right of the block give a hint of the computation order.

Figure 4.18 is the state diagram of the decoder. There are three loops in the right and two in the left. The "calculate gamma" in the block diagram works in the first loop, the block "calculate beta" works in the second loop, and the block "calculate alpha" and "calculate LLR" work in the third loop. The state of "wait" is inserted to avoid the data interference between the different stages. The states of "clear gamma", "clear beta", and "clear alpha", and "clear LLR" are inserted to reset the calculating blocks. The decoder with the sliding window technique works as following:

1. The 3×64 signals are received in the "Memory Data".
2. The signals from 31 to 0 are loaded as the first window. They are used calculate all branch metrics from the first stage, and the results are stored in the memory.
3. The backward metrics are calculated based on the branch metrics from the last stage,

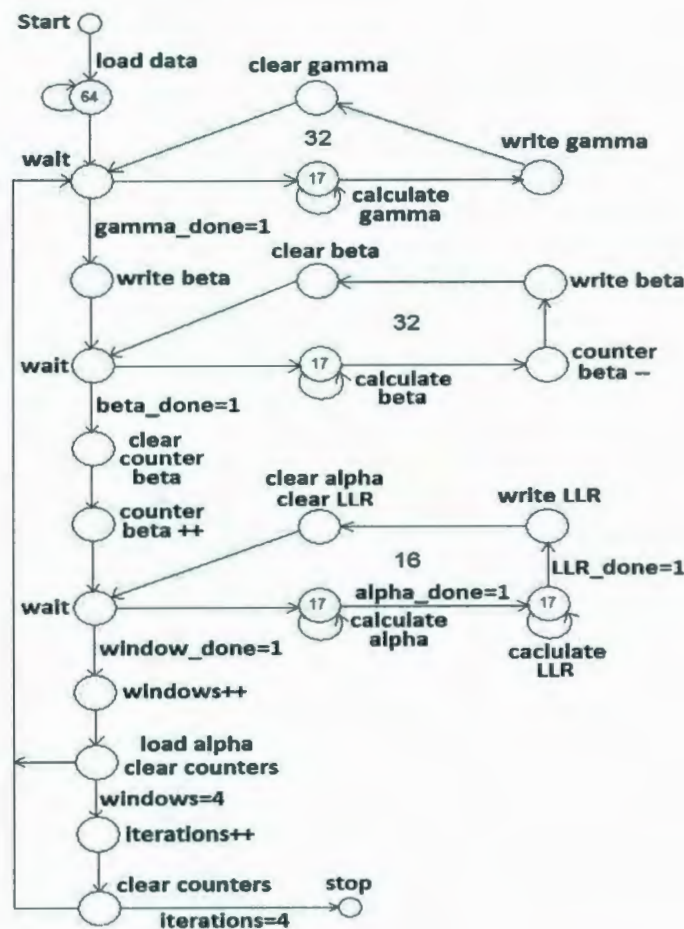


Figure 4.18: The state diagram of the turbo decoder

ceived signals and LLRs are loaded, every 32 stages as a window. Repeat step 2 to 7 to finish this half iteration.

9. Repeat step 8 until all half iterations are completed.

Obviously, the calculations of metrics are carried out serially according to the state diagram in Figure 4.18. The three loops on the right implement the MAX-LOG-MAP algorithm, the inner loop on the left implements the sliding window technique, and the outer loop on the left implements the iterative decoding. This is a straightforward design. On the right side, when any loop is working, the other two loops do nothing but wait. That means there is some inefficiency in the design, because of the waiting state, which can be further improved.

Note that the second loop and the third loop in the right are completely independent. But both of them depend on the first loop, the calculation of branch metrics. The block “Memory Gamma” keeps the branch metrics which are used in the second and third loops. We can make some modification in this implementation due to the relationships among these loops. We remove the block “Memory Gamma”, the largest memory that employs 32 RAMB18s, by calculating branch metrics again in the third loop. In this case, the first loop and the second loop is merged and a duplicated state of “calculate gamma” is inserted to the third loop. The block diagram is slightly modified. In Figure 4.17, the output of the block “calculate gamma” is connected directly to the blocks “calculate beta”, “calculate alpha”, and “calculate LLR”. The block “Memory Gamma” is removed and other blocks are unchanged.

Figure 4.19 illustrates this modified decoding process. The first loop in the right side illustrates that the backward metrics are calculated as soon as the branch metrics are available. The second loop in the right explains a similar process to obtain the LLRs. The branch metrics are calculated in both loops. Next, let us discuss the cost for this modification. After modification, the number of clock cycles in the first loop, which is $32 \times (17 + 17 + 3) = 1,184$, is less than $32 \times 20 + 32 \times 21 = 1312$, the sum clock cycles of the first and second loops in Figure 4.18. This is caused by the merges of the “overheads” states, such as “clear” and “wait” in these two loops. But the clock cycles in the second loop increases from $16 \times (17 + 17 + 3) = 592$ to $16 \times (17 + 17 + 17 + 3) = 864$. It is caused by insertion of the duplicated state of “calculate gamma”. When the window size is 32 and 16 bits are released, after the modification the total number of the clock cycles increases for every window from $32 \times 20 + 1 + 32 \times 21 + 2 + 16 \times 37 = 1907$ to $1 + 32 \times 37 + 2 + 16 \times 54 + 2 = 2,053$. If the block size is 1024, the window size is 32, and 16 bits are released in each window, there are 64 windows in a block. If the iteration number is fixed to 10, there are 20 half iterations. Then, the number for the increased clock cycles are $(2053 - 1907) \times 64 \times 20 = 186,880$ to decode such a block. This is the trade off between the memory size and the latency. However, we remove the largest memory of 32

RAMB18s, out of the total 51 RAMB18s, the saving rate is $32/51 = 62.7\%$, by calculating the branch metrics twice. So, this trade off is regarded as an improvement in efficiency.

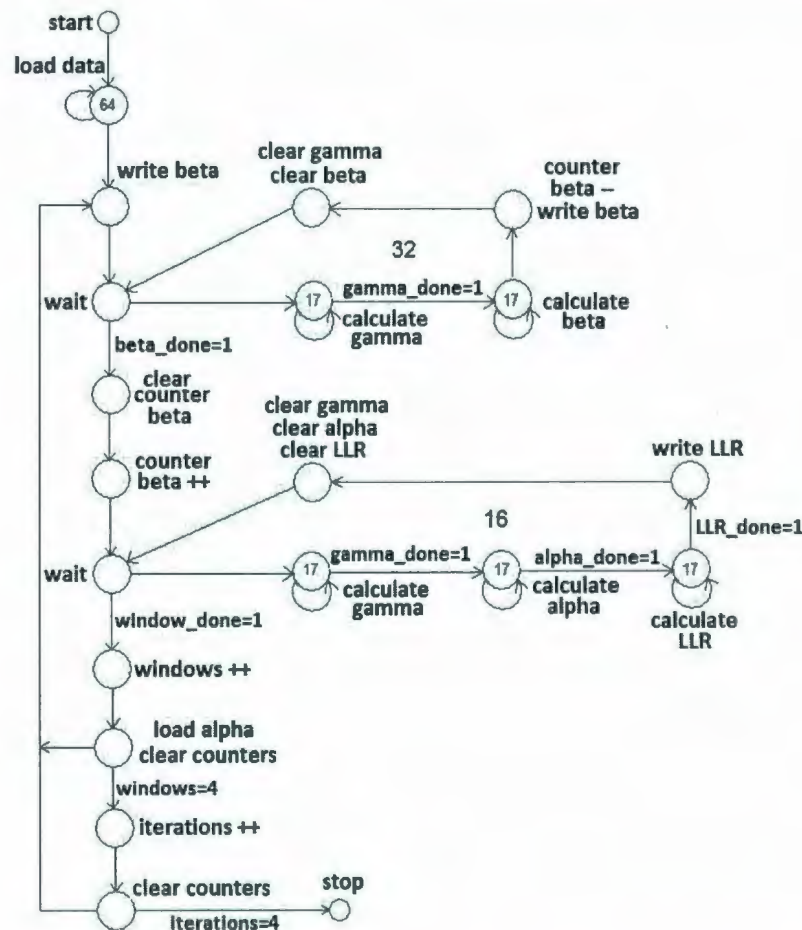


Figure 4.19: The state diagram of the modified turbo decoder

Figure 4.20 shows the decoded signals of two windows by Java simulations. The left is what is released from the first window in the first half iteration. The right is from the last window in the last half iteration. These data are used to compare with the VHDL waveforms, as shown in Figure 4.21, 4.22, and 4.23. In the waveforms, "wr_la" is the signal enable to write, "address_extrinsic" is the address used to write the LLR to the "Memory LLR", and "app" is the LLR released. It can be seen that from the simulation results the first 4 decoded signals, "0000", "0004", "0008", and "FFF6", perfectly coincide the VHDL waveforms. So does the first 8 decoded signals in the last window and last iteration. Thus,

the decoding correctness is verified by the VHDL waveforms.

The first 16 bits in the 1st window in 1st half iteration			The last 16 bits in the last window in the last half iteration		
File	Edit	Format View Help	File	Edit	Format View Help
0000000000000000	0000	0.0	0000000000001010	000a	2.5
0000000000000100	0004	1.0	1111111111111010	fffa	-1.5
0000000000001000	0008	2.0	0000000000010110	0016	5.5
1111111111110110	fff6	-2.5	0000000000011110	000e	3.5
0000000000001010	000a	2.5	1111111111100110	ffe6	-6.5
0000000000000000	0000	0.0	1111111111101000	ffe8	-6.0
0000000000000000	0000	0.0	0000000000010101	000a	2.5
0000000000000000	0000	0.0	0000000000011010	001a	6.5
0000000000001000	0008	2.0	0000000000011100	001c	7.0
1111111111111110	fffe	-0.5	0000000000010100	0014	5.0
0000000000001000	0008	2.0	0000000000000100	0004	1.0
1111111111111010	fffa	-1.5	0000000000000100	0004	1.0
0000000000001010	000a	2.5	1111111111111010	fffa	-1.5
1111111111110110	fff6	-2.5	0000000000000110	0006	1.5
1111111111110110	fff6	-2.5	0000000000000100	0004	1.0
0000000000000100	0004	1.0	1111111111110000	fff0	-4.0

Figure 4.20: Results of the turbo decoding in Java simulations

Figure 4.24 indicates that it takes about 3,347,000 ns to decode a block of 64 until the “done” signal is set to 1. It is actually 33,470 clock cycles because we use a clock of period 100 ns in VHDL simulations. We can also get this number from Figure 4.19. In the right, the first loop takes 1184 clock cycles. The second takes 864 clock cycles. There are 4 half iterations and 4 windows in the iterative decoding. We can approximately estimate the total clock cycles is more than $4 \times 4 \times (2048 + 5) + 64 = 32,912$. This number is approximate to 33,470 clock cycles which is indicated by the VHDL waveforms. Assuming the number of the turbo encoder state is 2^M , the window size is w , and in each window d bits are released, it can be calculated according to the Figure 4.19 that the decoding latency for such a window is approximate $5 + ((2^M + 1) + (2^M + 1) + 3) \times w + ((2^M + 1) + (2^M + 1) + (2^M + 1) + 3) \times d \simeq 2w \times 2^M + 3d \times 2^M = (2w + 3d) \times 2^M$. If the block size N is divisible by the window size w and the iteration number is fixed to i , then it takes approximately $(2w + 3d) \times 2^M \times N/w \times (2i)$ clock cycles to decode such a block with the sliding window.

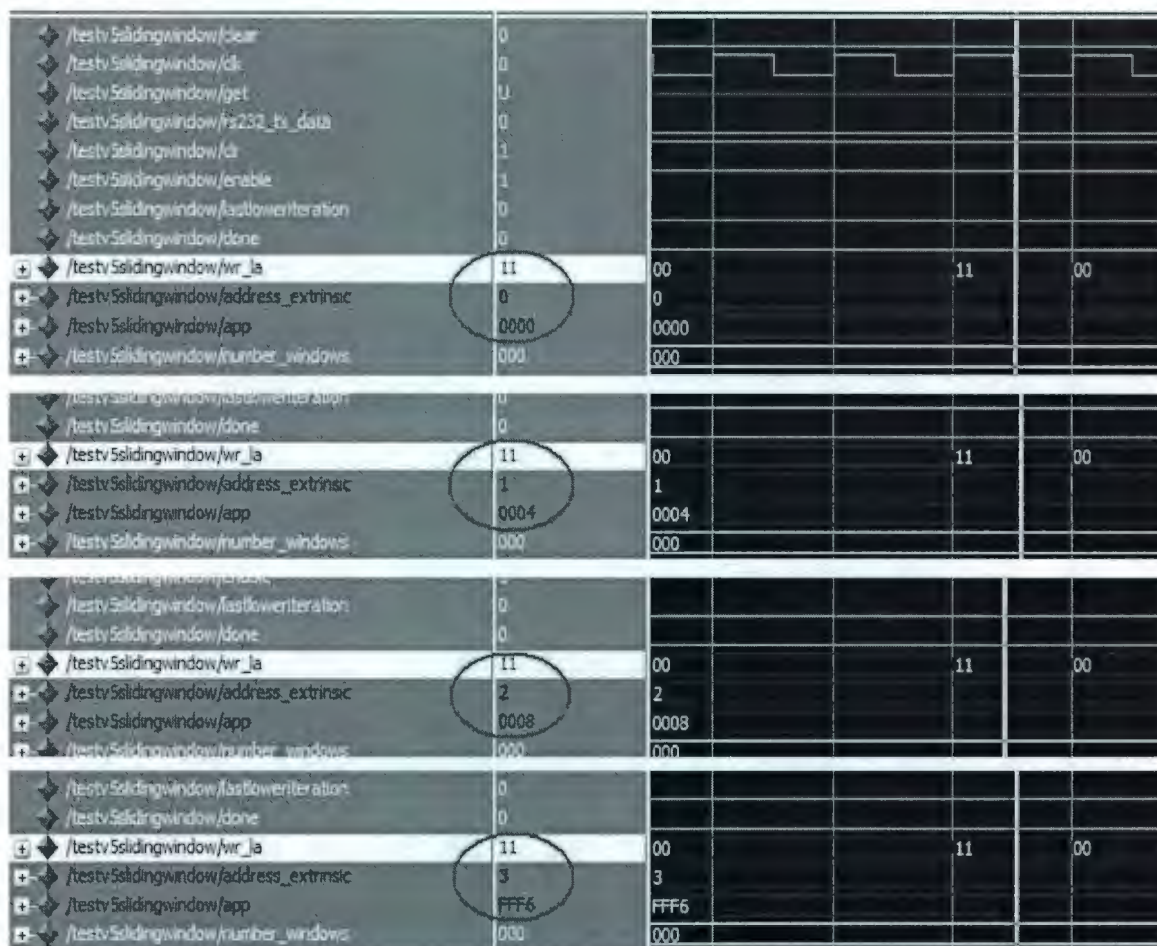


Figure 4.21: Waveforms (the first 4 bits in the 1st window) of the turbo decoder in VHDL implementation

4.3.3 The Implementation of a Pipelined Turbo Decoder

Although the VHDL implementation of the turbo decoder decodes the signals exactly as we have done in the software simulations, the decoding latency is comparatively long. It takes a large number of clock cycles, 2,051, to decode a window. How can we reduce this number of clock cycles? The answer is to pipeline the computing. In computing, a pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are executed in parallel or in time-sliced fashion [30]. For example, instruction pipelines, such as the classic *reduced instruction set computer* (RISC) pipeline, which are used in processors to



Figure 4.22: Waveforms (the decoded bit 0-3 of the last window) of the turbo decoder in VHDL implementation

allow overlapping execution of multiple instructions with the same circuitry. The circuitry is usually divided up into stages, including instruction decoding, arithmetic, and register fetching stages, wherein each stage processes one instruction at a time. In a pipeline, the multiple processes cooperate and work in parallel.

When we study the MAX-LOG-MAP algorithm in Figure 4.19, we realize that in two loops the calculation of metrics takes more than 90% of clock cycles ($32/35 = 91.4\%$, $48/51 = 94.1\%$). In either loop, there are more than one states of “calculate”. The “calculate” states are next to each other. Furthermore, the clock cycles to complete each “calculate” state are exactly 17, which means the calculating blocks have the same computing length in time. We apply the pipelining in these loops. The block diagram of a pipelined

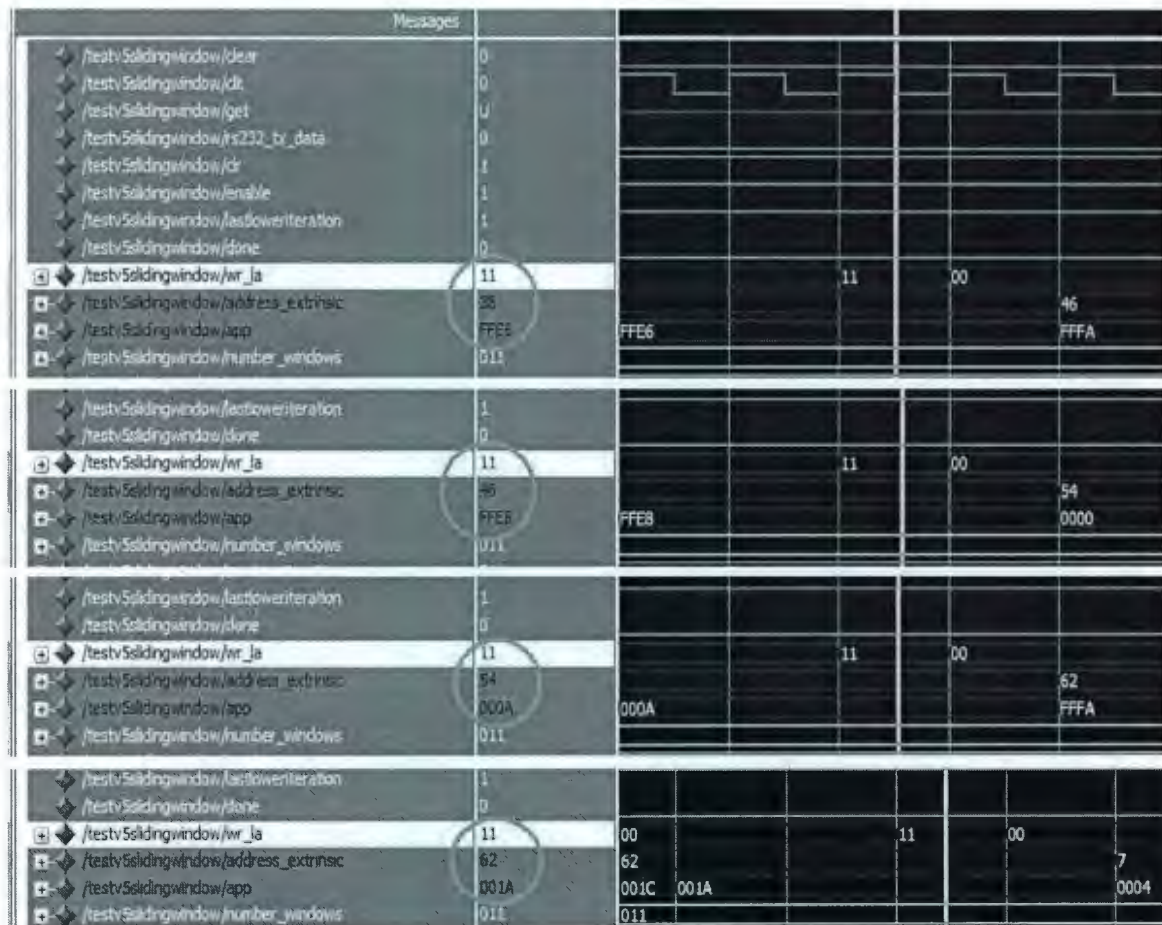


Figure 4.23: Waveforms (the decoded bit 4-7 of the last window) of the turbo decoder in VHDL implementation

turbo decoder shown in Figure 4.25 is close to the previous modified decoder. There is extra pipeline registers “register gamma” used to keep the branch metrics when the block “calculate gamma” is working pipelined. The size of registers is exact $2 \times 32 \times 16 = 1,024$ bits, because there are 32 branch metrics with 16 bits of each. The branch metrics for the extrinsic and a posteriori probabilities used in the last half iteration are different, so this number is doubled. This pipeline register is regarded as the cost for the pipelining. Furthermore, the controller is more complex.

The implementation of pipelining in this turbo decoder is a kind of request/acknowledge system. This request/acknowledge system is used to coordinate the different processes and avoid the data interference. There are two processes to calculate the metrics in the MAX-

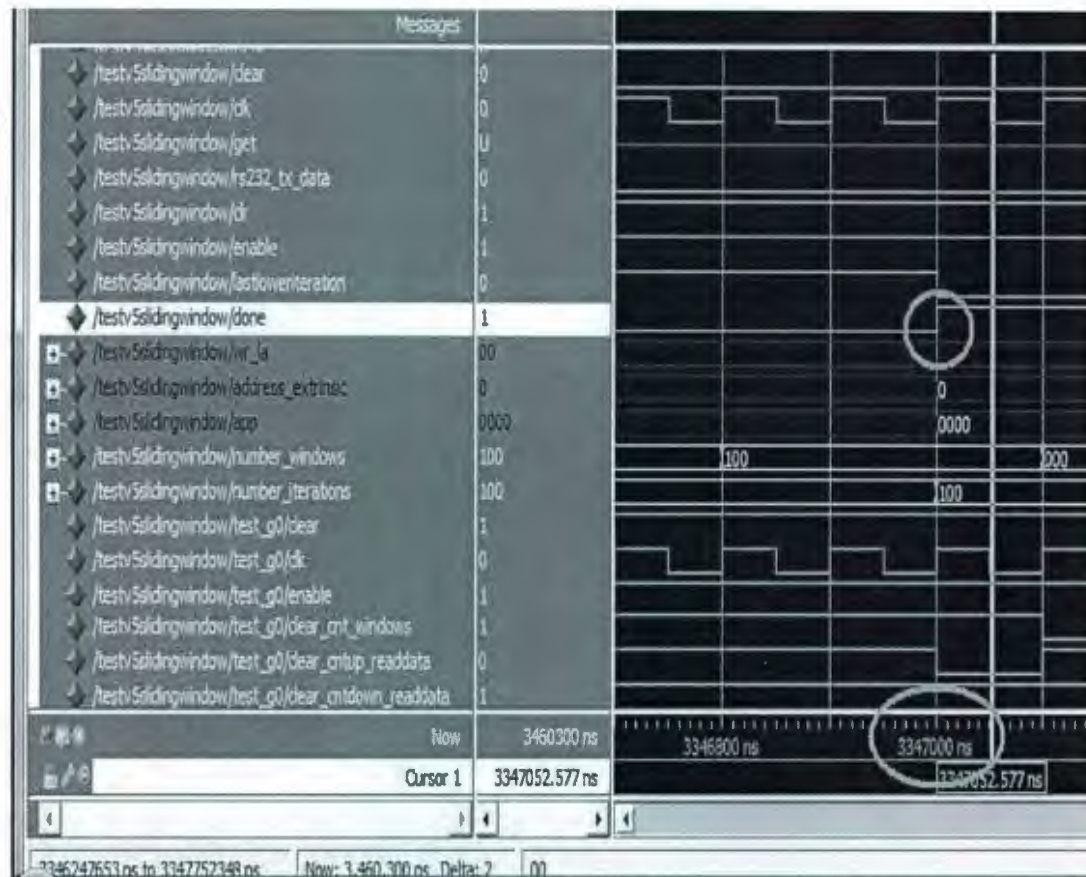


Figure 4.24: The decoding time of the turbo decoder

LOG-MAP algorithm. In a stage in trellis diagram, one process send a request and wait for the acknowledgement from the partner. After receiving the acknowledgement from its partner, this process enters into the next stage to calculate the metrics. The two processes cooperate as two threads in parallelism. Figure 4.26 shows the timing sequence of the two calculating processes, where “R” denotes the request, “W” denotes the data writing, and “A” denotes the acknowledgement. For simplicity, we use the branch (process A) and backward metrics (process B) to explain the interaction between them. Note that the backward metrics depend on the branch metrics and process A is always leading. That is to say, the time to write the branch metrics to the pipeline registers is critical to calculate the backward metrics.

The top part of Figure 4.26 shows the timing for the decoder without pipelining. The branch and backward metrics are calculated alternately. When one process is working, the

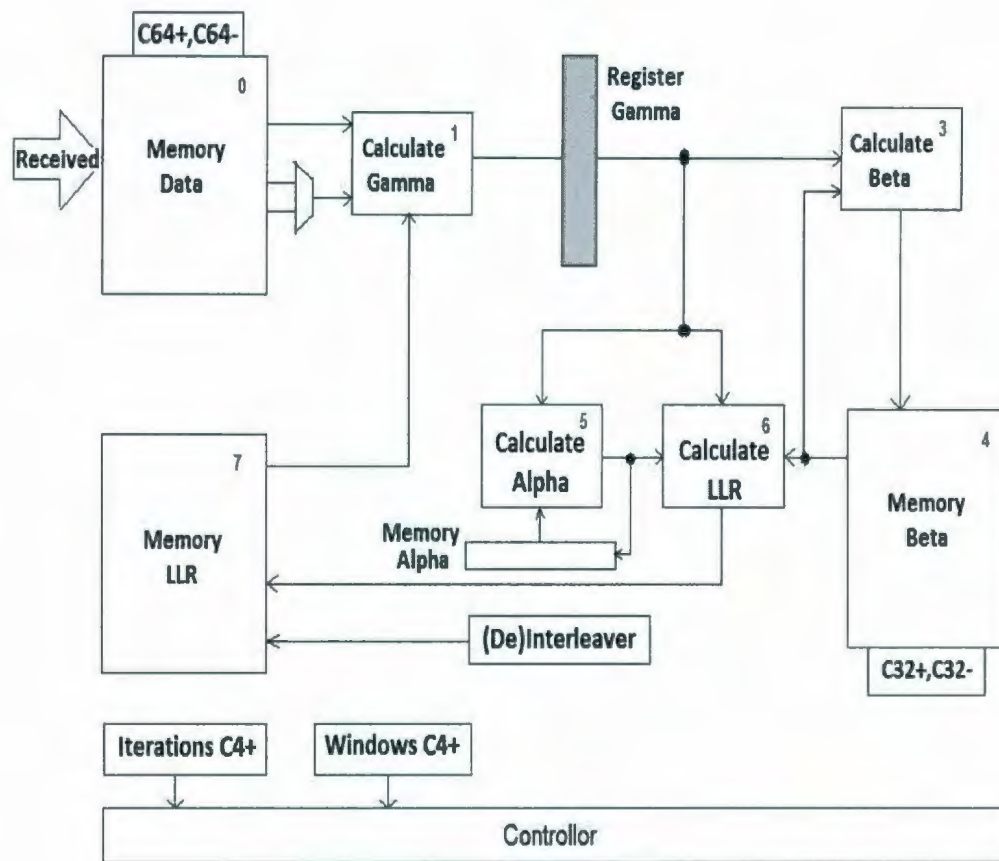
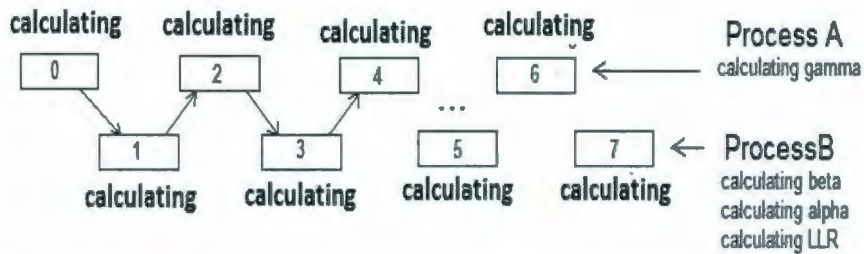


Figure 4.25: The block diagram of a pipelined turbo decoder with a sliding window

other is idle until the partner has complete this stage. All work is done serially, without improving the efficiency. The lower part of this figure shows the pipeline timing. The calculations of branch or backward metrics are concatenated by its own "A", "W", and "R" segments. However, the most important improvement is that the calculations of branch and backward metrics are in parallel. One process writes the calculating results and enters into the next stage as soon as it gets the acknowledgement from its partner. Once a process is busy calculating, it cannot be interrupted by the partner's request and acknowledgement. In this scheme, the decoder works as following:

1. For the process A: The 2×16 branch metrics in the first stage are calculated. Since it is the first stage, the acknowledgement B (from the process B) is always true. Then branch metrics are written to the pipeline registers. The acknowledgement A, which means the branch metrics for this stage is available, is released. Without any

Without pipelining



With pipelining

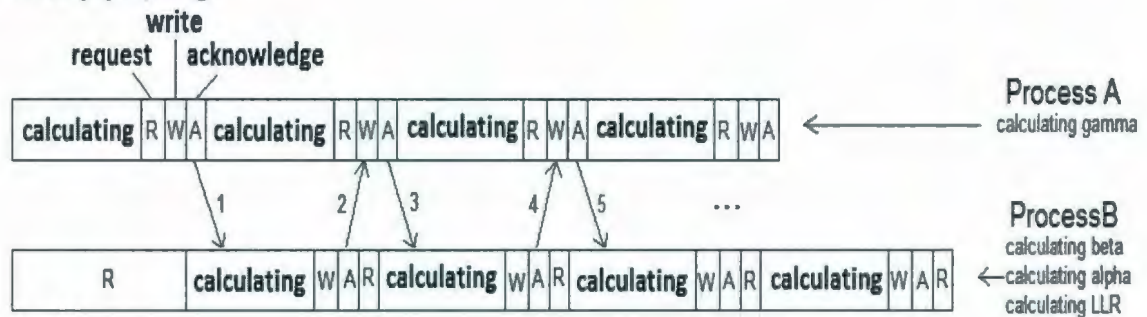


Figure 4.26: The timing in the MAX-LOG-MAP algorithm

delay, process A continues to calculate the branch metrics in the second stage. It will complete the calculation. It stops to wait before writing the data, with sending a request to the process B.

2. For the process B: At the very beginning, process B is waiting. As soon as it receives an acknowledgement A (from the process A), the process B starts to work (as shown by arrow 1). After the calculation, the results are written to the memory. Then the process B releases an acknowledgement B, which means the branch metrics in this stage is no longer used. The process B is now sending a request and waiting for the next acknowledgement A.
3. For the process A: After it gets acknowledgement B, it writes the data to the pipeline registers (as shown by arrow 2). It sends an acknowledgement A, then continues to calculate the branch metrics in the next stage without interruption. It stops before the writing as in step 1.
4. Repeat step 2 and step 3 alternately, until all backward metrics are calculated.

The processes interaction to calculate the forward metrics and LLR are the same as the above steps. This request/acknowledge system is designed to prevent the data out of boundary between stages. Note that the states “calculate gamma” and “calculate beta” in Figure 4.19 take the fixed number, 17, of clock cycles. It is possible to arrange the parallel computing without the request/acknowledge system, because the process A and process B have the same number of clock cycles in calculation. The data interference will not happen if it is deliberately designed. However, we apply the request/acknowledge system because of data safety in a general implementation. We should confirm the decoding correctness first while we were not absolutely confident about the calculating length in time. Second, the pipeline design is adaptive to the cases in which the calculating lengths in time of processes are unbalanced.

In this pipeline strategy, the decoding latency will be significantly reduced at the cost of the pipeline registers and a more complex controller. The controller has to coordinate the parallel computing, including generating request and acknowledgement signals. It is complex for a single controller to supervise all three processes. Therefore, the controller is broken down to three small controllers. The main one is the controller that is used to supervise the process of “calculating gamma”. The remaining two controllers are used to supervise the processes of “calculating beta” and “calculating alpha and LLR”. The three controllers exchange the messages to carry out the request/acknowledge system. Figure 4.27, 4.28, and 4.29 illustrate the state diagrams of three controllers, where “Ack_gamma”, “Ack_beta”, and “Ack_alpha” denote acknowledgements, the letters of “A”, “W”, and “R” in the states denote the segments “A”, “W”, and “R” in Figure 4.26. Similar to the FSM above, the loops in the right in the main controller complete the MAX-LOG-MAP algorithm, and the loops in the left complete the iterative decoding and the sliding window technique. Some “wait” states inserted between the loops that are marked by “R” are employed as the barriers to prevent the data interference. All three controllers begin at the common state “start”, but only the main controller ends at the state “end”. After this end state, other two controllers is continue waiting for the acknowledgement. But they are

never triggered again.

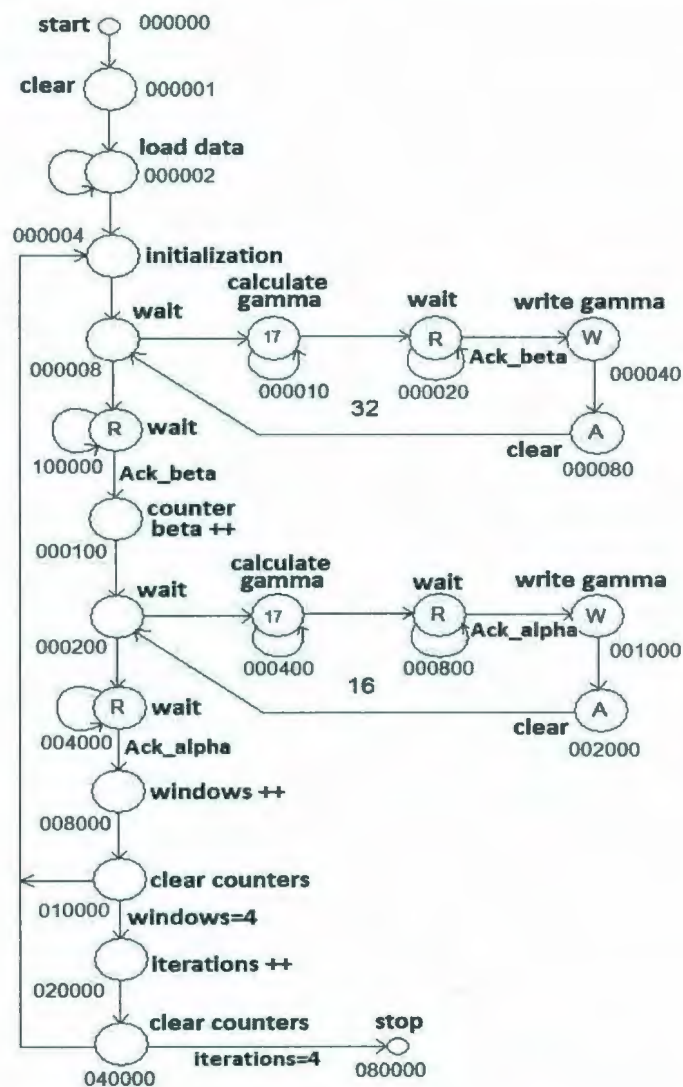


Figure 4.27: The main controller in the pipelined turbo decoder

Here we do not repeat the software simulation results and VHDL waveforms, which are identical to those in Figure 4.20, Figure 4.21, Figure 4.22, and Figure 4.23. When pipelining is employed, the VHDL waveforms also verify the software simulations. Furthermore, the output signals from the FPGA board, transmitted through an RS232 cable to a computer and then displayed in a hyper-terminal window, provide the evidence of decoding correctness in the FPGA implementation, as shown in Figure 4.30. It can be seen that the last 16 decoded signals in the last window, which are underlined in the hyper-terminal

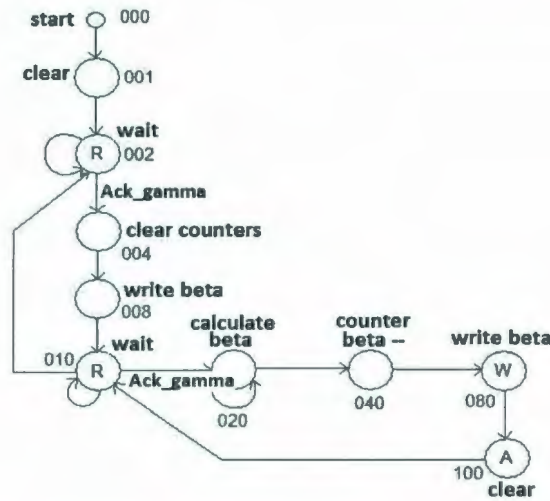


Figure 4.28: The beta controller in the pipelined turbo decoder

window, are identical to those in Figure 4.20. Actually, the output signals of the modified turbo decoder, which is discussed in the last section, are successfully displayed in a hyper-terminal window to verify the FPGA implementation. Therefore, we conclude that the pipelined decoding architectures in both VHDL and FPGA are exactly the same as that in software simulations.

However, what we are more interested in, is the reduction of the decoding latency. If the latency is represented by clock cycles, how many clock cycles we may save when employing the pipelining? Since the main controller works throughout the decoding process, its working period is regarded as the decoding latency. In Figure 4.27, the number of clock cycles in the states of "R" cannot be predicted, because it depends on the acknowledgement of other controllers. But it is expected to be a few clock cycles. If this "R" state is longer than 17 clock cycles, it almost makes no sense to apply the pipelining. We can read it from the waveforms, as shown in Figure 4.31 and 4.32. The state "000020" lasts for two clock cycles, and the state "100000" lasts for $(79100 - 77300)/100 = 18$ clock cycles. In Figure 4.27, for the first loop in the right, it takes $(1 + 17 + 2 + 1 + 1) \times 32 = 704$ clock cycles, except the last stage in a window. For the second loop in the right, it takes $(1 + 17 + 2 + 1 + 1) \times 16 = 352$ clock cycles, except the first stage in a window. In the

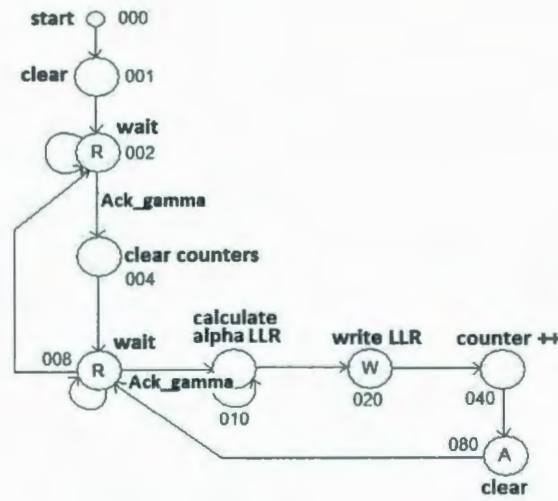


Figure 4.29: The alpha controller in the pipelined turbo decoder

first window, the state “000020” and “100000” take only one clock cycle. The first and the second loops have the same structure. Therefore, we can count the number of clock cycles in this pipelined decoder. For a window of 32, it takes $((1 + 17 + 2 + 1 + 1) \times 32 + 18 + 1) + ((1 + 17 + 2 + 1 + 1) \times 16 + 18 + 1) = 1,094$. Therefore, to decode a block of 64 needs approximately $4 \times 4 \times 1094 + 64 = 17,568$ clock cycles, which is very close to the decoding time, 17,627 clock cycles, in the VHDL waveforms, as shown in Figure 4.33.

Note that in the last section the modified turbo decoder that decodes a block of 64 needs approximately $4 \times 4 \times (1184 + 864 + 5) + 64 = 32,912$ clock cycles. However, the pipelined turbo decoding latency can be estimated approximately as 17,568 clock cycles. Thus, the improvement rate by the pipeline is about $(32912 - 17568)/32912 = 46.6\%$. It is achieved at the cost of a 1024-bit pipeline register. This decoding latency, in terms of clock cycles, is further estimated for a general turbo code. Assuming the number of the turbo encoder state is 2^M , the block size is N , the number of iterations is fixed to k , the window size is w , and in each window d bits are released, the pipelined decoding latency for a window is approximate $w(2^M + 1 + 5) + d(2^M + 1 + 5) + 40 \simeq w \times 2^M + d \times 2^M$ clock cycles. If we take the half window release strategy, $w = 2d$, then the rate of reduced clock cycles is about $4/7$. In general, comparing with the modified turbo decoding latency, the clock cycles

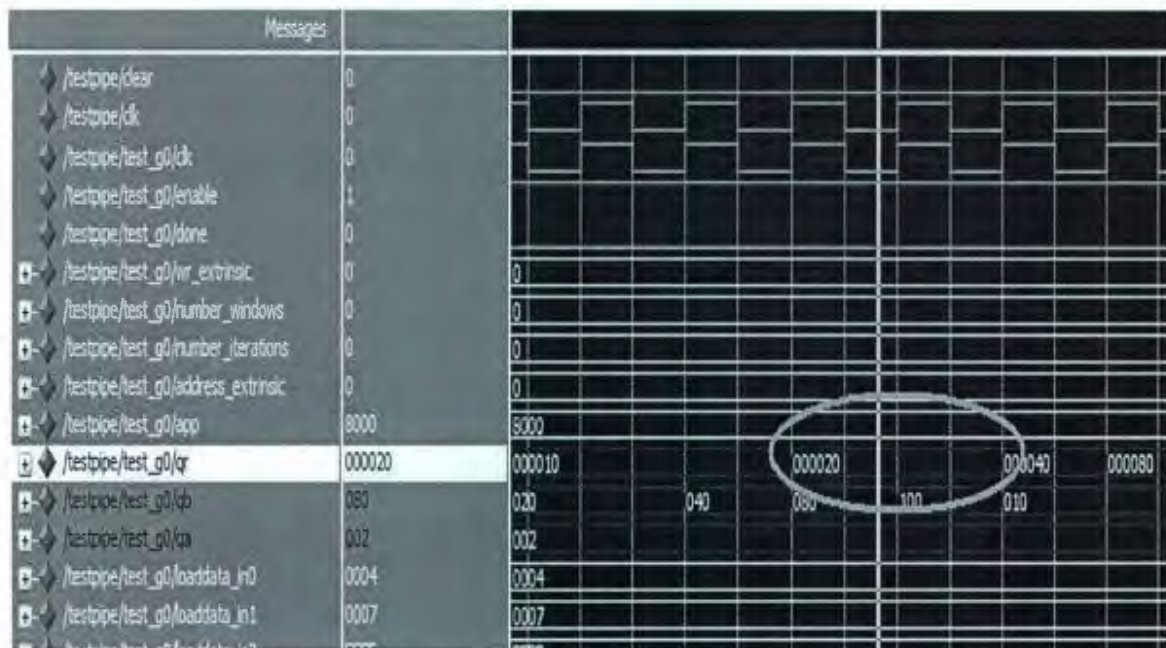


Figure 4.31: The state of “000020”

compare the waveforms and software simulation results. The hardware decoding correctness and the software model are verified by each other. The decoding performance in terms of BER for the hardware implementation and software simulation are completely identical. That means, it is reliable to use a Monte Carlo method in the software simulations, instead of hardware implementation, to investigate the decoding performance at BER.

Second, from the last section, it can be seen that there are trade-offs between the hardware size and decoding efficiency. However, the pipelining strategy provides a significant improvement in decoding latency, at cost of a small group of pipeline registers, 1,024 bits. In Table 4.1 we list the number of main primitives employed and the clock cycles used in decoding a window of 32 and a block of 64. Obviously, the pipeline is the best strategy, which saves 2/3 RAMB18s and about 1/2 clock cycles. It is a significant improvement in decoding efficiency.

Third, we compare the hardware complexity with the Xilinx 3GPP turbo decoders implemented on Virtex-5 and Virtex-6 family. In [31], an 8-state turbo decoder, which is used in 3GPP, are implemented in both Virtex-5 and Virtex-6 FPGA boards. Xilinx

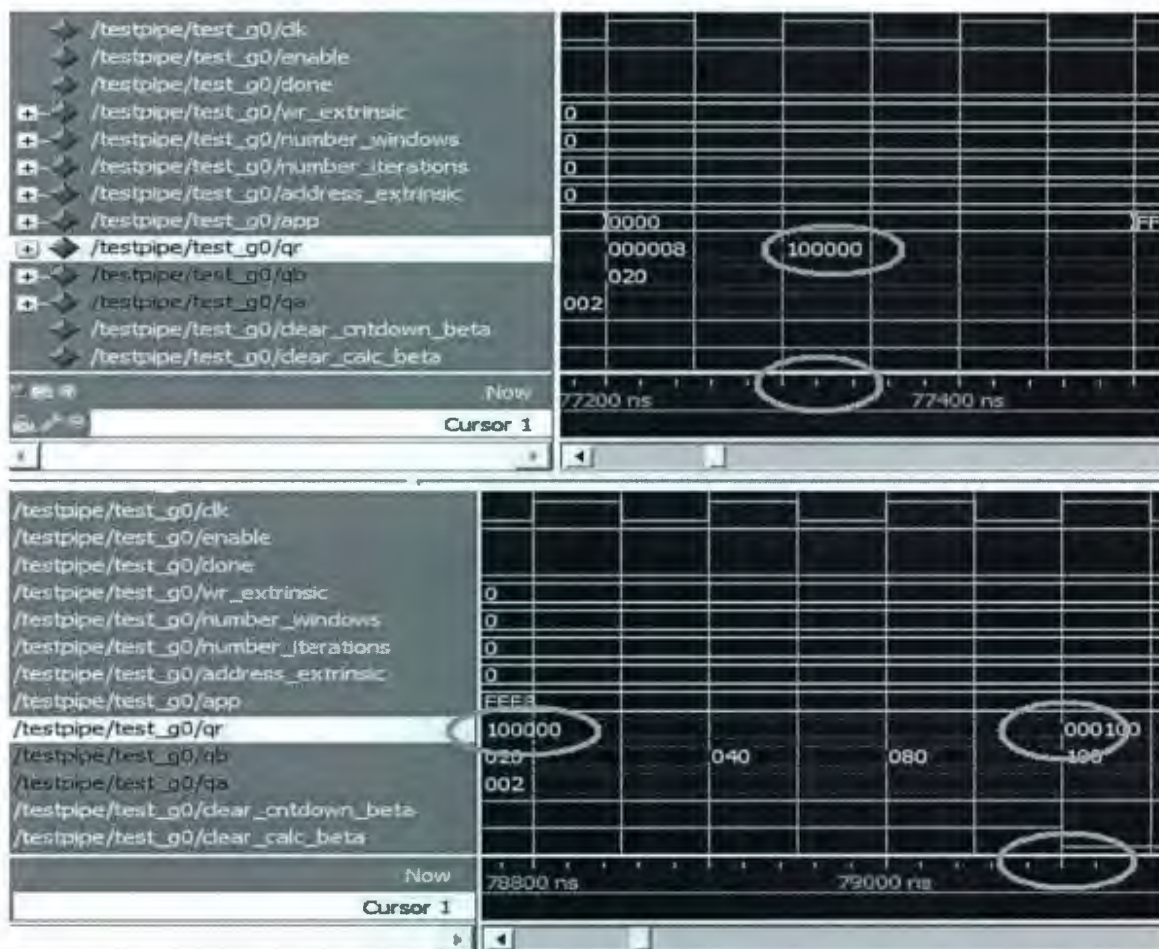


Figure 4.32: The state of “100000”

provides the resource requirements and decoding performance of this 3GPP turbo decoder. Here we mainly discuss the resource requirements. The numbers of the gates, flip-flops, and registers are not predicted unless the implementation is at gate level, because the gates, flip-flops, and registers are automatically optimized by the software. Some main primitives, such as the main memory blocks and DSP slices, can be designated by the hardware designer. Since the memory is the largest component in the turbo decoder, to compare the memory gives us a basic idea of the hardware complexity. Figure 4.35 illustrates the resource requirements for the different decoders. The left two columns are the implementations on Virtex-5, the middle two columns are the implementations on Virtex-6, and the most right column is the implementation of the pipelined decoder. A



Figure 4.33: Decoding time of the pipelined turbo decoder

pair of RAMB18s are equal to one RAMB36. It can be seen that the number of memory blocks are very close to each other. The implementations on Virtex-6 need $13 \times 36\text{kB}$ memory, while others need $10 \times 36\text{kB}$ memory. The number of multipliers used in the 3GPP decoders are not provided. We employ two DSP slices as two embedded multipliers, because the multiplications are indispensable. The numbers of *look up table* (LUT) pairs and slice LUTs in the 3GPP decoders on the Virtex-5, are close to those in the pipelined decoder. The implementation on Virtex-6 takes fewer LUT pairs and slice LUTs, about $1/4$ less. The number of slice registers in the pipelined decoder is about $1/4$ smaller than that in the 3GPP decoders. From the above, the hardware complexity of the pipelined decoder is close to that of 3GPP decoders. However, the 3GPP decoder has 8 states while the pipelined decoder has 16 states. Therefore, it can be inferred that the implementation of the pipelined decoder is acceptable in terms of hardware complexity.

Fourth, we consider the requirements for the pipeline technique. A cost of 1024-bit

Device Utilization Summary				
Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	3,290	69,120	4%	
Number used as Flip Flops	3,290			
Number of Slice LUTs	3,298	69,120	4%	
Number used as logic	3,292	69,120	4%	
Number using O6 output only	3,202			
Number using O5 output only	2			
Number using O5 and O6	88			
Number used as exclusive route-thru	6			
Number of route-thrus	8	138,240	1%	
Number using O6 output only	7			
Number using O5 and O6	1			
Number of occupied Slices	1,407	17,280	8%	
Number of occupied SLICEMs	0	4,480	0%	
Number of LUT Flip Flop pairs used	4,931			
Number with an unused Flip Flop	1,641	4,931	33%	
Number with an unused LUT	1,633	4,931	33%	
Number of fully used LUT-FF pairs	1,657	4,931	33%	
Number of unique control sets	69			
Number of slice register sites lost to control set restrictions	34	69,120	1%	
Number of bonded IOBs	4	640	1%	
Number of BlockRAM/FIFO	13	148	8%	
Number using BlockRAM only	13			
Number of 18k BlockRAM used	20			
Total Memory used (KB)	360	5,328	6%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Number of DSP48Es	2	64	3%	
Average Fanout of Non-Clock Nets	3.87			

Figure 4.34: Device utilization summary of the pipelined turbo decoder

pipeline registers, which is used to isolate the branch metrics in different stages, is not expensive. Note that the parallel computing is always suitable to calculate the branch and backward metrics, when the backward metrics depends on the branch metrics and the branch metrics are independent on backward metrics. What are the requirements of an efficient pipelining decoding? From the state diagram in Figure 4.27, it can be seen that the “R” state inside the loops in the right, which is a waiting state, can be viewed as a cost of the pipelining. Minimizing the clock cycles in these states, for example, the state “000020”, improves the efficiency. The clock cycles in the waiting state are mainly

Table 4.1: Comparison of different decoding schemes of the MAX-LOG-MAP algorithm

	<i>Original</i>	<i>Modified</i>	<i>Pipelining</i>
Number of DSP48E	2	2	2
Number of RAMB18s	48	16	16
Number of Clock cycles for a window	1907	2503	1094
Number of Clock cycles for a block	30512	32912	17568
Other Costs	None	None	Registers(1024-bit)

	Core 1		Core 2		Core 3		Core 4		Pipeline Decoder
Options			lr,rd_output,ft_thres,sk				rd_output,ft_thres,sk		
Xilinx Part	XC5VLX50		XC5VLX50		XC6VLX75T		XC6VLX75T		XUPV5-LX110T
LUT/FF Pairs ⁽¹⁾	4717		4750		3765		3910		4931
Slice LUTs	3483		3790		3712		3858		3289
Slice Registers	4115		4146		4062		4099		3290
Block RAMs (36k)	6		6		6		6		0
Block RAMs (2x18k)	4		4		7		7		10
DSP Blocks	0		0		0		0		2 dsp48e
Speed Grade	-1	-3	-1	-3	-1	-3	-1	-3	-3
Max Clock Freq ^(1,2)	299 MHz	385 MHz	299 MHz	381 MHz	285 MHz	349 MHz	291 MHz	367 MHz	N/A

Figure 4.35: Comparison with the Xilinx turbo decoders

determined by the difference in the numbers of clock cycles between the state “000020” in Figure 4.27 and the state “020” in Figure 4.28, the difference in time between calculating branch metrics and calculating the backward metrics. The best strategy is to make them equal in the calculating time. Therefore, an efficient pipelining has some requirement on the metrics calculating time, for example, to calculate the branch and forward metrics in the same clock cycles although the number of the branch metrics is as twice as that of forward metrics.

Fifth, this pipelined turbo decoder prototype may be extended to build other turbo codes without much effort. The look up table that stores the trellis diagram may be updated to create a turbo decoder with different constituent RSC encoders. If the interleaver structure is changed, then the look-up table that stores the address mapping function should be employed or updated. To build a general pipelined decoder, if the number of

iterations is fixed to k , the counter "Iterations C4+" should be replaced by "Iterations $C(2K)+$ ". The counter for the window index should be replaced by "Windows $C(N/d)+$ ". If the interleaver size is " N ", then the counters " $C64+,C64-$ ", which are attached to the block "Memory Data", could be replaced by " $C(N)+,C(N)-$ ". For the same reason, the counters " $C(w)+,C(w)-$ " are applied, to address the block "Memory Beta", to configure the window size, and another counter " $C(d)+$ " is applied to configure the number of bits released in each window. Note that the memory block "Memory Beta" is utilized shallowly with a very low percentage, about $(w/1027)\%$. In a future design, this block, which is implemented by 16 RAMB18s, can be replaced by a smaller memory with size of $16 \times w \times 16$ bits. But it is not necessary to replace the block "Memory LLR" unless the interleaver size is larger than 1024. If the interleaver size is larger than 1024, we should employ a larger block memory primitive, for example, RAMB36. Furthermore, according to the limited bits to represent a number investigated in the last chapter, we may modify the 16-bit values to the 10-bit values, 9 bits for the integer part and 1 bit for the fraction bits. Therefore, this pipelined turbo decoder prototype can be configured and then extended to other turbo decoders.

4.4 Hardware Implementation in Cheaper FPGA Devices

When the resources utilization is considered this pipelined turbo decoder is implemented in other cheaper FPGA boards, because the powerful virtex 5 board provides abundant resources. These resources, including the memory blocks, are far more than necessary. The cheaper boards have fewer resources. The utilizations of cheaper FPGA devices become higher, and the cost of the hardware implementation drops.

This decoder is also developed in the Xilinx Nexys2 FPGA platforms, which includes an XC3S1200E chip in the Spartan3E family. The Xilinx Nexys2 platform is an economical FPGA device, which contains different primitives. When we implemented this decoder

in the Xilinx Nexys2 platform, the main primitives, RAMB18s and DSP48E slices, were replaced by RAMB16_S18_S18s and MULT18X18SIOs. Another difference is the on-board oscillator, which is 100Mhz in the Xilinx XUPV5-LX110T Evaluation Platform and 50Mhz in the Nexys2 platform, respectively. However, the implementations at both frequencies work without any problem. With the Xilinx Nexys2 FPGA platform, the decoding correctness is also confirmed. The decoding latency in terms of clock cycles remains unchanged, if the primitives are well configured. Furthermore, this pipelined decoder was also implemented in a smaller FPGA platform, "Spartan3E Starter", which contains an XC3S500E chip. This board is cheaper, and contains fewer gates and blocks of memory. We employed all 20 RAMB16_S18_S18s on board. When using "Spartan3E Starter", the utilization of "Slice Flip Flops" is 39%, the utilization of 4 input LUTs is 50%, and the utilization of RAMB16s is 100%. So far, it is the smallest FPGA device that we employed to implement the pipelined decoder.

4.5 Summary

A pipelined turbo decoder is proposed in this chapter. This decoder is implemented on a Virtex 5 board. The original design without pipelining is verified for decoding correctness. Then, the decoding algorithm is modified in the computational order to save memory blocks. The decoding delay is significantly reduced when pipelining is introduced. This delay is estimated in terms of clock cycles. The hardware complexity is compared with those of the Xilinx cores. Furthermore, the implementation of this pipelined turbo decoder is attempted in cheaper FPGA boards to improve the utilization of resources. Finally, the scalability of this pipelined turbo decoder is discussed.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Our study of turbo codes starts from concepts of error correction coding in information theory. Turbo codes are derived from convolutional codes, by introducing an interleaver between two parallel concatenated convolutional encoders, to spread the burst errors in the information sequence. They outperform linear block codes and convolutional codes, especially in a low SNR environment. After turbo codes were proposed in 1993, they were widely applied in the communications, for example, in satellite communications and planetary probes. Although turbo codes have a disadvantage of a very high computational complexity that originally prevented them from implementation in hardware, recent VLSI development makes it easier to employ the turbo decoding in more applications.

We explain the architectures of encoders and decoders in detail. For the encoders, the constituent RSC encoders and the interleaver determine the free distance asymptotic performance. For the decoders, the SOVA/MAP algorithm or their variants are generally employed as the suboptimal decoding algorithm. These algorithms calculate the likelihood ratios, while the iterative decoding requires the exchange of likelihood ratios between the different SISO units. Finally, the likelihood ratios in the last iteration are used to make

a hard decision, to release whether the information bit is "1" or "0". Furthermore, we briefly explore the codeword weight properties, which significantly influence the decoding performance of turbo codes. From the performance union bound of the convolutional codes, the free distance asymptotic performance, which is used to illustrate the error floor in turbo codes, is derived.

Turbo decoding performance is mainly evaluated in terms of BER when running simulations in software. Due to the uncertainty of the information bits, some randomness is introduced in turbo codes. Therefore, Monte Carlo methods are employed to repeat simulations for a very large number of the information bits. We investigate two 16-state turbo codes, and show the decoding performance on simulation results. The free distance asymptotic performance is also provided to compare with the decoding performance. Actually, they are close to each other in medium and high SNRs in both two samples, so that their decoding performance can be approximately estimated from the free distance asymptotic performance.

Here we have considered two essential issues. The first is the sliding window technique, which is used to separate a large block into many of small windows. It helps to reduce the hardware complexity, because each time the SISO unit only decodes a window, not a block. The cost of the sliding window is some decoding degradation in performance. That's the trade off. The second consideration is the minimum number of bits used to represent numbers. This consideration is critical in the hardware implementation, whatever for the memories and arithmetic units. Too many bits used for numbers means redundancy, while not enough bits cannot confirm the decoding correctness. We determine the minimum number of bits to represent numbers through Monte Carlo simulations in software. Finally, we found that 9 integer bits and 1 fraction bit are sufficient for the investigated turbo decoder, without causing significant decoding performance degradation.

Hardware implementation is further used to verify the decoding correctness in the software simulations. We complete this implementation in both VHDL simulation and FPGA implementation. The verification strategy is to develop the hardware design, compare the

waveforms with the software simulation results, confirm the decoding correctness, improve the decoding efficiency, and further reduce the hardware complexity. A 16-state turbo decoder prototype is implemented on a Xilinx FPGA platform. The MAX-LOG-MAP algorithm is implemented first. Then computation process is modified to save a majority of memory blocks by calculating the branch metrics twice. In the modified decoder, the computation order is changed due to the sliding window technique. To further improve the decoding efficiency, pipelining, which is a popular concept in computer architectures, is introduced. A pipelined turbo decoder is developed. With pipelining, a pipeline register is inserted and then parallel computing of branch metrics and other metrics is realized. This pipelining strategy significantly shortens the decoding latency in terms of clock cycles. For example, about 46% of clock cycles are saved when the pipelining strategy is employed, if the window size is 32 and in each window 16 bits are released.

The following are the main contributions in this thesis. First, the software simulations provide an overall survey of turbo decoding performance. Two hardware implementation considerations, the sliding window technique and limited bits representing for numbers, are investigated in software simulations. These investigations provide the basis for the further FPGA implementation. Second, a 16-state turbo decoder is implemented in VHDL and finally programmed in different Xilinx FPGA devices. The software simulations, the VHDL codes, and the FPGA design are verified for decoding correctness by each other. The FPGA design is flexible. It can be configured, by changing counters and/or replacing look up tables, to construct other specified turbo decoders. Third, the relationship between the metrics in the decoding algorithm is observed. The calculation of branch metrics is as twice as that of forward and backward metrics individually. The calculation of branch metrics is independent on the forward and backward metrics, while the calculation of forward and backward metrics depends on the branch metrics. Thus, a pipelining technique with a request/acknowledgement system is introduced to improve the computational efficiency. This technique significantly reduces the decoding latency.

5.2 Future Work

First, further work should be done on this hardware implementation with a block of a larger size, although this implementation does not need much modification if the interleaver size is less than 1024. The scalability of this implementation is discussed, but not tested and verified. More work need to do in the extension of this prototype. Second, the longest path determines the clock skew. The design working in an over high frequency may generate the decoding errors in terms of BER. Future work can also be done to find out the max frequency to optimize this design. Third, a look up table can be added to update the MAX-LOG-MAP algorithm to the LOG-MAP algorithm. Fourth, the software simulations may help us to find turbo codes with one of the best decoding performance. For example, to replace block interleavers by random interleavers. This work will make a critical move to apply turbo codes in real applications. Fifth, the turbo codes with a higher memory order may be investigated and then implemented, when the pipelining technique is employed.

References

- [1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon Limit Error-correcting Coding and Decoding: Turbo Codes," in *Proceedings of IEEE International Conference Communications (ICC'93)*, pp. 1064-1070, May 1993.
- [2] S. Lin and D.J. Costello, *Error Control Coding*, Prentice Hall, 2004.
- [3] R. G. Gallager. *Low-density Parity-check Codes* [on-line]. Available: www.rle.mit.edu/rgallager/documents/ldpc.pdf [Mar. 13, 2010].
- [4] 3rd Generation Partnership Project, "Multiplexing and Channel Coding(FDD)," *3G TS 25.212*, June, 1999.
- [5] B. Vucetic and J. Yuan, *Turbo Codes, Principles and Applications*, Kluwer Academic Publishers, 2000.
- [6] D. Tse and P. Viswanakh, *Fundamentals of Wireless Communication*, Cambridge Uni. Press, 2005.
- [7] J. A. Briffa and V. Buttigieg, "Interleavers for Unpunctured Symmetric Turbo Codes," in *2000 Proceedings of IEEE International Symposium on Information Theory*, pp. 450, June 2005.
- [8] D. Divsalar and F. Pollara, "Turbo Codes for PCS Applications," in *Proceedings of IEEE International Conference Communications (ICC'95)*, Seattle, WA, pp. 54-59, June. 1995.
- [9] Communication Research Centre Canada, *Turbo8 - an 8-State Turbo Code Design and Simulation Tool*. [on-line]. Available: www.crc.ca/en/html/fec/home/turbo8/turbo8 [Jan. 28, 2010].

- [10] Consultative Committee for Space Data Systems, "Recommendations for Space Data Systems, Telemetry Channel Coding," *BLUE BOOK*, Oct., 2002.
- [11] S. Crozier and P. Guinand, "High-performance Low-memory Interleaver Banks for Turbo Codes," in *Proceedings of VTC'2001-Fall*, vol. 4, pp.2394-2398, 2001.
- [12] J. Gwak, S. K. Shin, and H. M. Kim, "Reduced Complexity Sliding Window BCJR Decoding Algorithms for Turbo Codes," in *Proceedings of the 7th IMA International Conference on Cryptography and Coding*, pp.179-184, 1999.
- [13] S. A. Barbulescu, "Sliding Window and Interleaver Design," in *Electronics Letter*, vol. 37, pp.1299-1300, Oct. 2001.
- [14] Y. Nawaz, "Reduced Complexity Turbo Decoders", Master's Thesis, Memorial University of Newfoundland, 2003.
- [15] D. Zhang, J. Liu, and X. Yang, "MAP Decoding Algorithm with Sliding Window for UST-symbol Based Turbo Code," in *2008. 11th IEEE Singapore International Conference on Communication Systems*, pp.674-678, Nov. 2008.
- [16] S. B. Wicker, *Error Control System, for Digital Communication and Storage*, Prentice Hall, 1995.
- [17] L. C. Perez, J. Seghers and D. J. Costello, "A Distance Spectrum Interpretation of Turbo Codes," in *IEEE Transactions on Information Theory*, pp. 1698-1709, Nov., 1996.
- [18] J. Seghers, "Final Rep., Diploma Project SS 1995, No. 6613," *Swiss Federal Institute of Technology, Zurich, Switzerland*, Aug. 1995.
- [19] Y. Wu, B. D. Woerner, and T. K. Blankenship, "Data Width Requirements in SISO Decoding with Modulo Normalization," *IEEE Transactions on Communications*, pp. 1861-1868, Nov. 2001.
- [20] G. Montorsi and S. Benedetto, "Design of Fixed-point Iterative Decoders for Concatenated Codes with Interleavers," *IEEE Journal on Selected Areas in Communications*, pp. 871-882, May. 2001.

- [21] T. Yanhui, T.H. Yeap, J.Y. Chouinard, "VHDL Implementation of a Turbo Decoder with LOG-MAP-based Iterative Decoding," *IEEE Transactions on Instrumentation and Measurement*, vol. 53, pp. 1268 - 1278, Aug. 2004.
- [22] Xilinx, *Xilinx University Program XUPV5-LX110T Development System* [on-line]. Available: <http://www.xilinx.com/univ/xupv5-lx110t.htm> [Mar 13, 2010].
- [23] Xilinx (2009, Nov. 5). *Virtex-5 FPGA User Guide*. [on-line]. Available: www.xilinx.com/support/documentation/user_guides/ug190.pdf [Jan 10, 2010].
- [24] Xilinx (2009, Jan. 12). *Virtex-5 FPGA XtremeDSP Design Considerations*. [on-line]. Available: www.xilinx.com/support/documentation/user_guides/ug193.pdf [Jan 10, 2010].
- [25] L. Inkyu, "Modification of the MAP Algorithm for Memory Savings," *IEEE Transactions On Signal Processing*, vol. 53, pp. 1147-1150, Mar. 2005.
- [26] R.R. Marin, A.D.G. Garcia, L.F.G. Perez, J.E.G. Villarruel, "Hardware Architecture of MAP Algorithm for Turbo Codes Implemented in A FPGA," in *Proceedings of the 15th International Conference on Electronics, Communications and Computers*, pp. 70-75, Feb. 2005.
- [27] G. Masera, G. Piccinini, M.R. Roch, and M. Zamboni, "VLSI Architectures for Turbo Codes," *IEEE Transactions on VLSI systems*, vol. 7, pp. 369-379, Sept. 1999.
- [28] Xilinx (2009, Feb. 6). *Virtex-5 Family Overview*. [on-line]. Available: www.xilinx.com/support/documentation/data_sheets/ds100.pdf [Jan 10, 2010].
- [29] A.M. Cortes (2008), "M.Sc. Thesis: VLSI Architectures for Turbo Codes Adapted the Standard Mobile Communications WCDMA(3GPP TS 25.212)," [on-line]. Available: freedownloadbooks.net/Thesis+-VLSI-pdf.html [Jan 10, 2010].
- [30] Wikipedia, *Pipeline Computing*. [on-line]. Available: en.wikipedia.org/wiki/Pipelining [Jan 10, 2010].

- [31] Xilinx (2009, Jun. 24). *3GPP Turbo Decoder v4.0*. [on-line]. Available: www.xilinx.com/support/documentation/ip_documentation/tcc_decoder_3gpp_ds318.pdf [Jan 10, 2010].
- [32] E. Boutillon, W.J. Gross, and P. G. Gulak, "VLSI Architectures for the MAP Algorithm," *IEEE Transactions on Communications*, vol. 51, pp. 175-185, Feb. 2003.



